A RELATIONAL APPROACH TO THE AUTOMATIC GENERATION OF SEQUENTIAL SPARSE MATRIX CODES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Paul Vinson Stodghill

August 1997

A RELATIONAL APPROACH TO THE AUTOMATIC GENERATION OF SEQUENTIAL SPARSE MATRIX CODES

Paul Vinson Stodghill, Ph.D.
Cornell University 1997

This thesis presents techniques for automatically generating sparse codes from dense matrix algorithms through a process called *sparse compilation*. We will start by recognizing that sparse computations are ubiquitous to scientific computation, that these codes are difficult to write by hand, and that they are difficult for conventional compilers to optimize. We will present the sparse compiler as an alternative to writing these codes by hand or using sparse libraries.

We will show how many aspects of sparse compilation can be modeled in terms of relational database concepts, These include the following: queries to express sparse computations, relations to model sparse matrices, the join operation to model simultaneous efficient access of sparse matrices. Using this model, the problem of sparse compilation can be seen as an instance of the query optimization problem.

We will discuss two basic strategies for sparse compilation based upon this relational approach. One strategy is targeted towards algorithms that can be described using inner join queries, which include matrix-vector multiplication and matrix-matrix multiplication. This approach is the one that we have currently implemented. The other can handle a larger class of dependence-free matrix algorithms. Although it is more general, the latter approach introduced does not generate as efficient code for some problems as the former approach. We will show that these two approaches are grounded in properties of the relational algebra and draw connections with previous work that has been described in the database literature. We also discuss how conventional dense optimizations and fill can be handled within the overall relational framework.

We will discuss the Bernoulli Sparse Compiler and use experimental results to show that this system is able to generate sparse implementations

from non-trivial dense matrix algorithms that are as efficient as hand-written codes. In addition, this compiler provides a novel mechanism that allows the user to extend its repertoire of sparse matrix storage formats. Thus, the user is not only able to choose the data structures for storing the sparse matrices, but to describe these data structures as well.

BIOGRAPHICAL SKETCH

Paul Vinson Stodghill was born in Providence, Rhode Island, and grew up in Carlisle, Pennsylvania. His formative years were spent pulling things apart and not putting them back together, simultaneously discovering the joy of music at loud volume, reading too much, and verbally bludgeoning his opponents while a member of his high school forensic debate team. These were skills that he found very useful in college.

He attended Dickinson College, which was close enough to home that his parents could still hear his stereo, and from which he received a B.A. in mathematics and computer science and with Latin honors in May, 1988. He then entered graduate school in the Department of Computer Science at Cornell University, from which he received a M.S. in May, 1992 and a Ph.D. in August, 1997.

He plans to live happily every after.

To Cindy and Christopher, Mom and Dad.

ACKNOWLEDGEMENTS

I am indebted to Dr. Keshav Pingali, who served, not just as my advisor, but also as my role model. His enthusiasm for teaching and pursuit of simplicity in research have provided me with many valuable lessons. He has been a great source of wisdom, and I feel privileged to have worked under him.

I also thank the other members of my committee, Adam Bojanczyk, Tom Coleman, and Rich Zippel, all of whom provided valuable comments and suggestions that helped greatly in steering the research and preparing this dissertation.

I also wish to acknowledge Vladimir Kotlyar, who was a wonderful co-worker and an invaluable source of knowledge. This and his wonderful sense of humor have made working with Vladimir one of the most productive partnerships that I have had the pleasure of being part of.

I would like to thank Dr. Nancy Baxter for her inspiration. Her supervision of my undergraduate senior project helped kindle an interest in programming languages and compilers that I will carry with me throughout life.

There are many people here at Cornell and in Ithaca who helped me along the way. My fellow students have provided me with many wonderful friendships and other amusing diversions. I want to thank all of the people with whom I played hockey and ultimate Frisbee. James Allan, Alessandro Panconesi, Vladimir Kotlyar, Jonathan Russell-Anneli, and Pan Chen were all wonderful roommates.

The people in the department who provide administrative and computing support have been very, very helpful. They have made this learning and research environment a very effective and pleasant place to be.

I especially want to thank my parents. My mother, Beverly Mullen Stodghill, never lost her faith in me and was always compassionate and supportive. Having been through graduate school himself, my father, Dr.

Jack Stodghill, was extremely understanding and provided me with excellent advice for getting through in one piece. I have a tremendous amount of respect for my father and the quiet example of excellence in teaching that he provides.

Children can teach us more about life and what makes it important than can be learned in the classroom or by doing research. Christopher Hubbell has been a wonderful teacher, sharing with me the excitement of a turtle laying eggs, or the discovery of a fossils in a quarry, or his enormous curiosity in most anything he is involved in, but mostly his joy and enthusiasm. I hope that one day he will be as fortunate as I am.

Cynthia Anne Robinson is my partner and the love of my life. Her example of effectiveness served as an inspiration, and her patience and support made this endeavor possible. She and Christopher were my reasons to keep pushing, and words cannot adequately express how thankful I am to her for everything that she has given me. With great anticipation, I look forward to moving on with our lives together.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Part I

Preliminaries

Chapter 1

Introduction

1.1 Exploiting sparsity in scientific computation

A program developed for performing scientific computation on high performance architectures should, ideally, run as efficiently as possible. By reducing the amount of memory required to store the program, a scientist is able to simulate larger and larger problems and thus is able to see macro phenomena instead of micro phenomena. By reducing the amount of time required to perform the calculations, the scientist is able to solve larger problems more accurately in the same amount of time. Thus, reducing memory and time requirements allows the scientist to perform bigger and better simulations.

Parallelization, the process of dividing the space and time requirements of a computation over many processors, is one of the most important ways of increasing the performance of scientific codes. However, parallelization is not the focus of this thesis. High performance on parallel machine is, of course, obtained by maximizing parallelism *and* sequential performance. In this thesis, we will focus on maximizing sequential performance.[1]

In terms of sequential performance, one of the most important things that can be done to reduce memory and time requirements in scientific computation is to exploit *sparsity* wherever possible. In the context of matrix computations, sparsity refers to zero matrix entries, and a matrix is said to

[1]We will, however, briefly discuss how these sequential techniques support and compliment parallelization in Section 18.3.3.

be sparse if a large percentage of its entries are zeros. Matrices having less than 1% of non-zero entries are not uncommon in these sorts of codes.

1.2 Classifying sparse codes

There are many different ways to characterize sparse codes. Perhaps two of the most important properties are whether they contain do-any loop nests, and whether they create fill.

A loop nest is said to be a *do-any loop nest* if its iterations can be executed in any order without affecting the correctness of the computation. A compiler has the freedom to scheduling the iterations of a do-any loop nest for maximum performance without affecting the correctness of the computation.[2] Matrix-Vector Multiplication (MVM), is an instance of a do-any loop nest, since the order in which the updates to y may occur in any order. Backward triangular solve is not a do-any loop nest since the iterations that perform the updates to a location in the solution vector must be performed before that location is scaled.

When performing a computation on a sparse matrix, the situation may arise when a zero value in the matrix becomes non-zero. This can happen, for instance, when factoring a sparse matrix. The update operations of a factorization will sometimes create non-zeros, which are called *fill*. When fill occurs, the data structure used to represent the sparse matrix must be updated to include these new non-zero entries.

In Figure 1.1, we assign various computations to categories depending upon whether or not they are do-any loop nests and whether or not they have to handle fill (MMA = Matrix-Matrix Addition, MMM = Matrix-Matrix Multiplication).

Class I. The computations in Class I have the property that the do not involve loop carried dependencies or the creation of fill. With these restrictions, one might guess that the computations in this class are trivial, and are quite simple when written using FORTRAN or MATLAB. However, this simplicity masks two important aspects of these codes.

[2] We ignore the possibility of roundoff errors introduced by transformations performed by the compiler. This is a standard simplification in the compiler literature.

		do-any? Yes	do-any? No
fill?	No	I: MVM, MMA*, MMM*	III: Forward and Backward Solve
fill?	Yes	II: MVM, MMA†, MMM†	IV: Matrix Factorization

* If the result of the computation is dense.

† If the result is sparse.

Figure 1.1: A classification of sparse codes

The first aspect is that they very prevalent in sparse matrix computations. MVM (shown in Figure 1.2(a)), in particular, is one of the core computations of many indirect, or iterative, solvers. The Krylov Space methods ([108], [111], [15], [60]), of which Conjugate Gradient (CG) ([66], [113]), GMRES ([110]), and the Lanczos method ([88]) are perhaps the most well known, are examples of this type of solver. At this heart of these computations is the repeated multiplication of the linear system by residual vectors. Such methods are becoming increasingly popular because, in practice, they often require fewer computational resources than other solution techniques ([72]).

Class II. The computations in Class II allow the creation of fill but not loop carried dependencies. There are two primary instances of these computations in practice. The first, which is called *matrix assembly*, occurs in Finite Element ([115], [91]) and other methods when many extremely sparse matrices are added together to form a single linear system. A code to perform matrix assembly is shown in Figure 1.2(b) and is a special case of the more general MMA computation.

Another less frequently occuring instance of this class of computation is sparse MMM ([14]), which is shown in Figure 1.2(c). This computation can occur in certain formulations ([49], [50]) of the Multigrid method ([115], [28]).

Class III. The computations in Class III allow loop carried dependencies but not fill. Some of the most frequently occuring computations in this

```
for i := 1 to n do
    for j := 1 to n do
        Y[i] := Y[i] + A[i,j] * X[j];
    end do
end do
```

(a) MVM

```
for e := 1 to num_elems do
    for i := 1 to n do
        for j := 1 to n do
            A[i,j] := A[i,j] + E[e,i,j];
        end do
    end do
end do
```

(b) Matrix assembly

```
for i := 1 to n do
    for j := 1 to n do
        for k := 1 to n do
            C[i,j] := C[i,j]+
                    A[i,k] * B[k,j];
        end do
    end do
end do
```

(c) MMM

```
X[1] := L[1,1]/B[1];
for i := 2 to n do
    X[i] := B[i];
    for j := 1 to i − 1 do
        X[i] := X[i] − L[i,j] * B[j];
    end do
    X[i] := X[i]/L[i,i];
end do
```

(d) Forward Triangular Solve

Figure 1.2: Some core computations

Figure 1.2: (Continued)

for $k := 1$ to n do
  $A[k,k] := \sqrt{A[k,k]}$;
  for $i := k+1$ to n do
    $A[i,k] := A[i,k]/A[k,k]$;
  end do
  for $j := k+1$ to n do
    for $i := j$ to n do
      $A[i,j] := A[i,j] - A[i,k] * A[j,k]$;
    end do
  end do
end do

(e) Cholesky

class are forward and backward triangular solves. The code for the forward triangular solve is shown in Figure 1.2(d). Preconditioning is a process in which an approximation of the inverse of a linear system is computed and used to speed up the convergence of iterative methods. This inverse is often computed using an incomplete factorization and triangular is used to apply the resulting triangular factors during each iteration of the method ([115], [111]). The emphasis of current research on sparse triangular solves is on parallel algorithms. In order to increase the amount of parallelism, these algorithms perform complex renumberings of the triangular factors and store them in sophisticated data structures that provide efficient access and allow good sequential performance ([8], [8], [3], [72]).

Class IV. The final, and perhaps most difficult, class of computations is Class IV, which allow fill and dependencies. Direct methods, or matrix factorization algorithms, are instances of these computations. The Cholesky factorization shown in Figure 1.2(e) is an instance of such a direct method. As with the triangular solves, complex matrix reordering and sophisticated data structures are required in order to obtain good performance on parallel computations. Supernodal, Multi-Frontal versions of the classic Cholesky ([59], [64], [117], [87], [52]), QR ([118]), and Least Squares ([119]) algorithms have recently been developed towards this end.

1.3 The process of exploiting sparsity

Very roughly, we can divide the process of exploiting sparsity in matrix computations into the three stages,

1. **The numerical algorithm is developed.** The abstract specification for a numerical technique is taken from a textbook or paper and its details flushed out. For instance, details like the preconditioner or convergence test, which may only be partially specified in the abstract algorithm, must be fully specified. A FORTRAN or MATLAB-like language can be used to specify the algorithm at this point. We call this representation of the algorithm its *dense specification.*

2. **Data structures for representing the sparse matrices are chosen** These data structures are chosen to complement the sparsity of

the physical problem, the algorithm, and the architecture on which the final implementation will be run. The aim is to minimize the amount of memory required to store the non-zero entries of the matrix while still providing a method for the algorithm to access the non-zeros efficiently. We call these data structures *sparse matrix storage formats*.

3. **The sparse implementation is produced** The numerical algorithm is specialized to take account for the selected storage formats to produce a final implementation. We call this the *sparse implementation* of the dense specification.

Example 1.1 Suppose that we want to compute a particular weighed sum of a sparse vector. We start by specifying the numerical algorithm to be performed,

$sum := 0;$
for $i := 1$ to n do
    $sum := sum + i * v[i];$

Then, we select a storage format for the sparse vector, v. Suppose that we decide to use two vectors to store the non-zero elements of v. If v has nz non-zero elements, then the nz length vector, $vind[i]$, will store the index of the ith non-zero of v, and the nz length vector, $vval[i]$, will store its value. This particular method of storing the non-zero entries of a sparse vector is called the *sparse vector* storage format.

Finally, we specialize the original dense specification into the following efficient sparse implementation,

$sum := 0;$
for $ii := 1$ to nz do
    $sum := sum + vvind[ii] * vval[ii];$

It is worth stressing the importance of exploiting the sparsity in this example: The original dense specification, if implemented directly, used n words of storage for the vector and performed n floating point operations (flops) in n iterations. The final sparse implementation code uses $2nz$ words of storage, and performs nz flops in nz iterations. If only 5% of the entries of v were non-zero, then the sparse implementation will use 1/20th of the space required by the dense specification and run in 1/20th of the time. When we derive sparse implementations of matrix computations that require n^2 space and n^2, or even n^3 time, the improvements can be even more dramatic.

A discussion of the decisions and techniques used in Stage 1 is beyond the scope of this thesis. Treatment of the material can be found in any numerical methods ([115]) or numerical analysis ([60]) textbooks. We simply assume that these decisions have been made and that a dense specification of the numerical algorithm is available. However, in order to understand the problem addressed by this thesis, we will discuss the decisions and transformations made in Stage 2 and 3 in greater detail.

1.4 Stage 2: Choosing a sparse storage format

One of the most important decisions made when transforming a dense specification into a sparse implementation is the choice of storage formats for the sparse matrices. There are three very general considerations when making this decision,

- Special properties of the matrix's sparsity,
- The computation being performed with the matrix, and
- The architecture on which the computations is to be performed.

We will discuss each in turn.

1.4.1 Properties of the sparsity

The more advanced storage formats presume certain properties about the sparsity of the matrix being stored. These formats provide efficient access and enable higher performance by exploiting these properties. The performance for sparse matrix-vector multiplication (MVM) written for different sorts of sparsity patterns and several different storage formats is shown in Table 1.1 and continued in Table 1.2.

Here is a brief explanation of data in Tables 1.1 and 1.2.

- Each row represents corresponds to a different sparse matrix. In Table 1.1, each combination of d, n, and c correspond to a different regular

Table 1.1: MVM for regular sparsity, in mflops

Grids			Mflops					
d	n	c	Diag.	Coor.	CRS	ITPACK	JDiag	B.S.
2	10	1	31.293	9.359	19.492	8.214	28.571	2.484
2	10	3	27.006	5.054	29.881	8.461	33.727	17.457
2	10	5	18.530	4.727	22.649	7.982	21.890	26.744
2	10	7	18.177	4.752	23.371	7.980	21.698	27.145
2	17	1	36.966	9.586	21.168	8.497	32.173	2.552
2	17	3	19.881	4.623	19.854	8.083	22.613	13.983
2	17	5	18.950	4.530	22.055	7.790	22.028	20.588
2	17	7	18.494	4.804	23.227	7.759	22.657	25.094
2	25	1	37.907	5.650	21.416	8.537	32.952	2.593
2	25	3	19.246	4.728	19.788	8.132	21.675	12.849
2	25	5	12.106	4.613	22.647	6.992	13.587	18.639
2	25	7	9.542	4.757	22.975	6.819	12.454	25.151
3	10	1	30.585	4.679	16.258	7.639	21.629	3.209
3	10	3	12.158	4.711	21.002	6.978	14.680	14.905
3	10	5	8.664	4.865	22.399	6.692	12.367	22.361
3	10	7	8.116	4.860	22.830	6.660	11.997	27.342
3	17	1	14.010	4.752	15.175	6.764	12.040	3.102
3	17	3	9.681	4.705	20.858	6.704	11.378	14.560
3	17	5	8.635	4.700	22.576	6.649	10.997	21.821
3	17	7	8.478	4.327	23.499	6.802	11.805	27.615
3	25	1	13.874	4.773	14.969	6.864	11.081	3.139
3	25	3	9.209	4.478	20.660	6.686	11.444	14.402
3	25	5	8.896	4.390	21.995	6.629	11.201	22.353

Table 1.2: MVM for various sparsity, in mflops

Grids	Mflops					
Name	Diag.	Coor.	CRS	ITPACK	JDiag	B.S.
tiny	2.222	1.695	1.754	1.852	2.128	0.403
small	21.972	8.595	16.000	7.446	21.818	2.921
medium	23.192	7.888	29.874	8.150	32.583	19.633
arco1	16.639	4.500	20.921	7.570	22.133	17.810
arco4	9.734	4.243	20.644	6.654	11.435	18.100
cfd.1.10	8.730	4.459	21.678	6.793	11.793	26.070
cfd.2.10	8.912	0.040	21.518	6.828	11.716	25.071
662_bus	0.671	6.275	17.145	5.176	32.256	1.885
685_bus	1.133	5.379	20.421	4.869	31.406	2.475
1138_bus	0.823	4.937	15.680	3.138	28.202	1.758
add20	1.513	4.536	16.404	1.104	19.593	4.407
add32	2.630	4.532	13.486	2.343	13.188	3.023
bcsstm27	15.130	4.807	23.677	7.714	21.604	28.907
bcsstk31	(1)	4.372	22.372	2.635	8.393	17.339
bcsstk32	(1)	4.243	23.383	3.041	10.804	23.078
e05r0000	8.534	4.841	26.642	5.188	25.085	(2)
gr_30_30	29.495	4.660	18.136	7.764	22.857	5.374
mahindas	1.963	4.606	16.603	1.635	19.830	3.215
memplus	0.268	4.648	15.299	0.250	12.111	4.138
nos4	12.719	9.245	20.952	8.071	29.700	3.201
nos5	5.822	4.879	23.403	5.757	28.693	5.685
nos6	38.658	5.441	20.634	8.551	32.945	2.570
nos7	35.749	4.836	20.000	8.131	27.830	3.259
orani678	2.492	4.799	23.628	0.642	20.971	(2)
sherman1	18.699	5.191	16.756	6.094	28.069	2.187

(1) Did not terminate in a reasonable amount of time.

(2) Sparsity pattern precluded the use of the BlockSolve format.

grid used to derive the sparse matrix. In Table 1.2, each "Name" corresponds to a different data set from various sparse matrix suites. The exact properties of the matrices used here and throughout this thesis are discussed in Appendix A.

- In order to make a comparison with dense storage formats, we have performed the MVM computation using the Diagonal Skyline storage. This format, described in Section 7.3.3, is a cross between a dense and a sparse storage format. The reason for including it is that it is "dense" enough that it will obtain the performance levels exhibited by other dense storage formats, while still having many of the beneficial properties of a sparse matrix format. It represents the best that one can do to minimize space while being essentially a dense storage format. The "Diag." column gives the performance in millions of floating point operations (mflops) of MVM performed using the Diagonal Skyline storage

- The last five columns give the performance in mflops of MVM performed using the Coordinate (Section 7.3.1), Compressed Row Storage (CRS) (Section 7.3.4), ITPACK (Section 7.3.5), Jagged Diagonal (Section 7.3.6), and BlockSolve (Section 7.3.7) storage formats, respectively.

- The machines and methodologies used for these results are described in Chapter 17.

Notice that there is not a single storage format that provided the best performance for all sparse matrices. This is because each matrix has different sparsity properties that are more appropriate for one format than the others. If the non-zeros of a matrix fall within a small number of diagonals, then dense storage formats are appropriate and will yield very high performance. If this is not the case and the mesh's sparsity is distributed more uniformly, then using any other sparse storage format will be an improvement. Furthermore, some of the sparse formats are more appropriate than Diagonal Skyline for some of the regular meshes. If a matrix is small or has few non-zeros in each row then a storage format, such as Jagged Diagonal, will be appropriate. This format collapses the non-zeros of the matrix into a few, very long, vectors. This approach will obtain the high performance, because it will have very long trip counts on its innermost loops. Suppose instead that a highly irregular spatial discretization using finite-elements is called for, but many components, or unknowns, are assigned to each grid point. In this case, dense submatrices can be found within the larger sparse matrix. For real problems, these dense submatrices can be quite large. The BlockSolve format is designed to store these dense submatrices in such a way that BLAS-2 and BLAS-3 routines can be used to perform the bulk of the MVM computation. Using machine specific versions of the BLAS libraries can yield very high performance in this case.

1.4.2 Properties of the Computation

In addition to accounting for the sparsity of the matrix, storage format selection must account for the algorithm that will use it. One trivial example of this point is matrix factorization. The update step of algorithms like LU or Cholesky may result in new non-zero entries being inserted into the sparse matrix. If the storage format selected for the sparse matrix does not easily accommodate new entries–none of the formats described in Section 7.3 do–then the resulting sparse implementation will likely be very inefficient.

More subtle effects can be observed in algorithms like backward triangular solve. This algorithm can be expressed in such a way that it traverses the sparse matrix either by rows or by columns. Storage formats like ITPACK and Jagged Diagonal provide access to the non-zeros within a row, so it is possible to generate a sparse implementation of this solver using these formats that is not horrendously inefficient. However, as we will see in Section 7.3.6, row access is not the prefered means of accessing these two storage formats. Rather, prefered access is along "diagonals" that do not directly correspond to either rows or columns in the original matrix. As a result, these storage formats will not achieve as high performance for triangular solve as a format like BlockSolve that is specifically designed for these algorithms.

1.4.3 Properties of the Architecture

Some of the storage formats described in Section 7.3 store the non-zeros of each row or column consecutively in memory. Depending upon the sparsity, there may be a relatively small number of non-zeros in each row or column, and this can result in short trip in the innermost loops of computations like MVM. On vector machines, short inner loop trip counts can lead to poor performance since the vector units will not fully utilized ([108]). Storage formats, such as ITPACK and Jagged Diagonal, circumvent this problem by storing non-zeros from different rows in very long vectors that are called “diagonals.” Because these diagonals yield very long inner loop trip counts, these storage formats yield much better performance on these sorts of architectures.

On modern RISC processors, the architectural feature that is perhaps the most important, and difficult, to manage well is the cache. Codes that exhibit a high degree of memory reuse will often perform better than codes that do not. Some storage formats, such as BlockSolve, attempt to organize the non-zeros of a matrix into small dense blocks. Then, codes such as MMM and triangular solve can be easily reorganized to exploit these dense blocks in order to increase memory reuse. These effects are much more difficult to achieve with storage formats that do not directly expose dense blocks.

1.5 Stage 3: Developing an efficient sparse implementation

In Stage 3, the dense specification of the algorithm developed in Stage 1 is specialized to exploit the sparsity present in the storage format chosen in Stage 2. There are many different tools and techniques that can be used in this process.

For instance, algebraic properties, like $x \cdot 0 = 0$, can be used to identify and eliminate useless computation. In the weighted sum example given above, the computation performed in the body of the dense specification is useless when $v[i]$ is 0. This is because the update to sum will also be 0. The sparse implementation ensures that the update to sum is not performed in this case.

Another consideration is ensuring that the sparse data structures are accessed efficiently. If a conventional dense array is used to store a matrix, then accessing any element of the matrix is relatively cheap; primitive array access is a constant time operation. However, if a more complex data structure is used to store the non-zero entries of a sparse matrix, then accessing arbitrary elements may be more expensive. Consider the sparse vector storage format described above: if an arbitrary element needs to accessed with a sparse vector, then a $\log nz$ binary search must be performed. However, the sparse implementation of the weighted sum code given above does not require such a search, because it traverses the data structure directly.

The weighted sum code was small and simple enough that an efficient sparse implementation could easily be generated. However, for larger and more complex numerical algorithms, there are many more transformations and optimizations that need to be performed in order to generate an efficient sparse implementation. Many users perform these steps themselves, but find the process very time-consuming and error-prone. It is very often the case that a dense, sequential numerical computation that can be described in tens of lines using MATLAB will take thousands or tens of thousands of lines of FORTRAN or C to code an efficient sparse, parallel implementation ([10], [73], [117]).

This is truly unfortunate because, not only is the initial cost of developing sparse numerical codes very high, the cost of changing from one algorithm or sparse matrix storage format to another can be just as expensive. As a result, once the user has developed a sparse implementation, they may be unwilling to make major modifications to it. Even though the changes might result in factors of speedup, these benefits may be outweighed by the added development cost.

For these reasons, there is a great need to remove the burden of developing efficient sparse implementations from the user's shoulders.

1.6 Sparse libraries

One approach to shifting the burden from the user is to implement libraries that contain commonly used functionality. This approach has been very successfully applied to dense computations, and libraries like BLAS ([89], [48], [47]) and LAPACK ([7]) are frequently used in scientific codes. Perhaps similar libraries could be developed for sparse computations.

1.6.1 Traditional Libraries

With traditional libraries, the library writer performs all three stages discussed in Section 1.3. With such libraries, the user is provided with sparse implementations of a set of algorithms implemented for a range of storage formats. The advantage of using these libraries is that the user's initial development cost is much lower; they only need to write code to interface with the libraries. As a result, the cost of migrating from one library to another that offers better performance is less daunting.

- SPARSKIT ([109]) provides many routines for performing linear algebra operations implemented for most of the sparse matrix storage formats listed in Section 7.3 and many others, as well. However, advanced and more recent storage formats, like BlockSolve, are not provided. Also, according to the documentation, most of the computation routines convert from each of these storage formats to a canonical format before performing the computation. In addition to incurring a large overhead from this conversion, as we saw in Section 1.4.1, this is simply not a good strategy: no single storage format achieves the best performance for all sparsity patterns.

- Several sparse extensions to the dense BLAS level 1, 2, and 3 routines have been proposed for handling sparse storage formats.

  Most dense BLAS function names have the form, XYYYZZ, where

  - X specifies the data type and is one of S = Single-precision real, D = Double-precision, etc.,
  - YY specifies the storage format and is one of GE = General, SY = Symmetric, etc., and
  - ZZ specifies the operation and is one of MM = Matrix-matrix product, SM = Solution of a triangular system with multiple right-hand-sides, etc.

  Argument conventions specify the arguments to each routine based upon the storage format and the operation. For instance, the convention for matrix-matrix product is

  XYYMM(TRANSA, TRANSB, M, N, K, ALPHA, $args(A)$, B, LDB, BETA, C, LDC)

  and the convention for expanding $args(A)$, when A is GE, is

  A, LDA

  so the arguments to the routine for multiplying two double precision General matrices is,

  DGEMM(TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)

In practice, the BLAS interface is not so well structured, but the sparse extensions work on the principle that it is.

One proposal ([31]) modifies these conventions by extending the storage specification to three letters, YYY, and by adding more storage names, like, COO = Coordinate, CSR = Compressed sparse row, etc. Appropriate conventions for expanding $args(A)$ for each of these new types are added as well.

Another proposal ([51]) adds a single new storage name CS = Compressed Storage whose expansion for $args(A)$ is,

FIDA, DESCRA, A, IA1, IA2, INFOA

FIDA is a string that specifies the storage format, like COO = Coordinate, CSR = Compressed sparse row, etc., and A, IA1, IA2, and INFOA contain the data needed to implement each format. The proposal provides a list of possible formats, but stresses that "this list is neither exhaustive nor necessarily supported in any particular implementation."

This disclaimer goes right to the heart of the problem with these libraries: it it is very difficult to standardize a list of sparse storage formats. Unlike the dense BLAS, in which a small handful of dense storage formats capture most of the cases used in practice, the set of sparse storage formats used in practice is simply too varied.

1.6.2 Object-Oriented Libraries

More recent libraries have adopted an object-oriented approach in order to circumvent this difficulty. In Stage 1 using this approach, the library writer implements the dense specification directly in an object-oriented language. In Stage 2, the user provides the implementation for a sparse matrix storage format in a form that satisfy the interface requirements of the library code. Stage 3 consists of linking the library's modules and the user's modules to produce the final sparse implementation.

- The PETSc library ([61],[10]) specifies a message-passing abstractions for sparse matrices. That is, there is an abstract matrix types that specifies operations for performing matrix-vector multiplication, and so on. PETSc also provides various implementations of this abstract matrix type. PETSc provides implementations for CRS as well as others. Entire libraries of iterative and non-linear solvers are built using this matrix abstraction. That way, these codes work for all of the various sparse matrix storage formats without modification.

While this method of implementation provides great flexibility for the PETSc designers (the BlockSolve storage format was integrated with little difficulty, for instance), the current implementation does not allow users of the PETSc library to add new storage formats without recompiling the whole library. This is an artifact of the fact the PETSc was implemented in C, which does not directly provide object-oriented programming constructs.

- Several more recent proposals ([46], [101], [100]) have suggest taking the same approach as PETSc but using C++ as the implementation language instead of C. The advantage of these libraries over PETSc would be that the user could provide additional sparse matrix storage formats without having to recompile the whole library. Apart from this, and some minor syntactic differences (like, using operator overload to provide a more "natural" interface to the user), the various C++ libraries provide a paradigm very similar to PETSc's.

Object-oriented libraries provide a much simpler interface than traditional libraries and have the additional benefit of extensibility. However, they still have several shortcomings that hinder aggressive compilation. First, the usual means of "packaging" libraries is as sets of precompiled modules. Thus, when the user's code is compiled, the library's code is not usually available. This prevents aggressive inlining of library code and subsequent optimization. Second, object-oriented constructs, such as dynamic method dispatch, often obfuscate the interprocedural control flow. Even with very sophisticated interprocedural analysis, it is often not possible to determine what particular method is being called at an invocation site. Analysis becomes impossible when the library source code is not available.

Several techniques have been developed for addressing these difficulties. One approach is to retain the original source code in some form, and to perform recompilation and optimization of frequently occuring method caller/callee pairs at run-time ([35], [9]). Because the original source is retained, high-level optimizations are possible. Because the run-time behavior of the program is used, method invocations can be correctly matched with the called method. The downside of this approach is that, because this is done at run-time, a delicate balance must be struck between speeding up the program by recompiling critical sections and spending so much time recompiling that these performance gains are lost. Optimizations for high-performance architectures, such as loop transformations for cache locality, are often very time-consuming and cannot be performed profitably in these circumstances.

1.6.3 Stage 3 policy decisions and optimizations

There is an even deeper problem that is shared by all existing libraries, traditional or object-oriented. When a programmer is implementing a library, they do not always know the context in which their code will eventually be used. As a result, they have to choose algorithms that work well in general but are not necessarily the best choices for all situations. Another problem is that sometimes the most efficient algorithms cannot be implemented behind a clean library interface.

```
-- Scatter the sparse y to the dense t
for ii_Y := 1 to nzs_Y do
    t[ind_Y[ii_Y]] := val_Y[ii_Y];
end do
-- Do the dot-product of x and t
sum := 0;
for ii_X := 1 to nzs_X do
    sum := sum + val_X[ii_X] * t[ind_X[ii_X]];
end do
```

Figure 1.3: Scatter dot-product

To illustrate these points, consider computing the dot-product of two sparse vectors. Two possible sparse implementations of this computation are shown in Figure 1.4 and the graph in Figure 1.5 shows the elapse running time of these implementations. The times were generated on an unloaded 50MHz Sun 670 MP. The lines marked “Scatter” and “Two-finger” show the

$sum := 0;$
$ii_X := 1; ii_Y := 1;$
while $ii_X \leq nzs_Y \wedge ii_Y \leq nzs_Y$ do
    if $ind_X[ii_X] = ind_Y[ii_Y]$ then
        $sum := sum + val_X[ii_X] * val_Y[ii_Y];$
        $ii_X := ii_X + 1; ii_Y := ii_Y + 1;$
    else if $ind_X[ii_X] < ind_Y[ii_Y]$ then
        $ii_X := ii_X + 1;$
    else
        $ii_Y := ii_Y + 1;$
    end if
end do

Figure 1.4: Two-finger dot-product

performance of the codes as they are shown in Figure 1.4. The line marked "Scatter (amortized)," shows the performance of the "Scatter" implementation in which only a fraction of the scatter loop cost is counted. This simulates the case in which the scatter loop can be hoisted outside of an enclosing loop and its cost can be amortized over many runs of the dot-product loop.

As we can see in Figure 1.5, the two-finger code out-performs the scatter code. For this reason, a library writer might choose to implement the two-finger version in a dot-product routine. However, if the scatter loop could be hoisted, then the scatter version out-performs the two-finger version. Of course, the library write has no way of knowing whether there is an enclosing loop in the user's code out of which the scatter can be hoisted. Furthermore, even if this were known, the library writer has no way of forcing the scatter to be hoisted into the user's code; current programming languages simply do not provide the library writer with a means of expressing this sort of optimization.

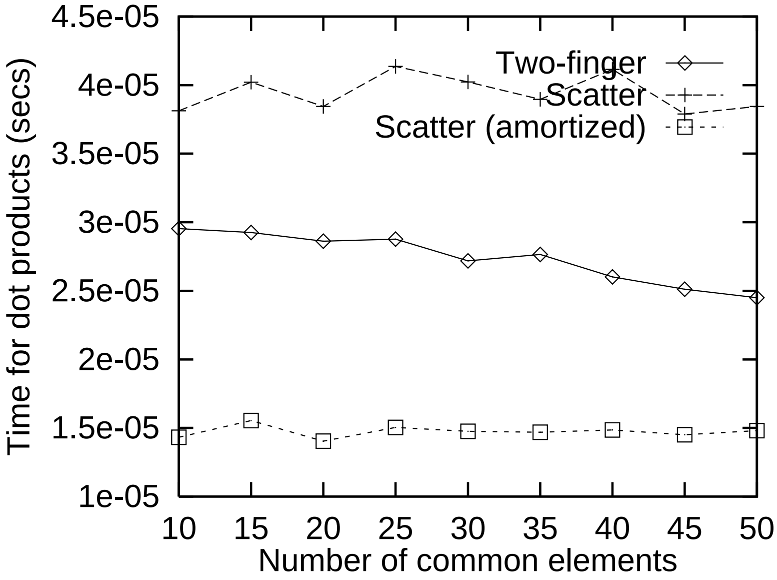


Figure 1.5: Various versions of sparse dot-product, in secs.

1.7 A better approach: the Sparse Compiler

We have observed that, when the user is writing sparse code by hand, they have complete control over policy decisions and optimizations that occur in Stages 2 and 3. However, exercising this level of control is difficult and tedious. Libraries are one approach to alleviating the user's burden, but often libraries do not give the user sufficient control over algorithmic and storage format decisions. Also, the natural abstraction layer between the user's code and the library code often serves as a barrier to important policy and optimization decisions. It seems that, instead of having the sparse implementation provided, a priori, what is needed is a tool that will generate portions of the sparse implementation from the dense specification.

We call such a tool a *sparse compiler*. A sparse compiler could quickly provide the user with sparse implementations, as do libraries, while still providing a mechanism by which important global implementation decisions and optimizations can be made, as does hand implementation.

In Chapter 2, we will describe a sparse compiler previously developed by Bik ([19]). We present this previous work at a high level, stressing its goals and giving an overview of its design. In Chapter 4, we will introduce our approach to sparse compiling, focusing on the high-level aspects of our work and contrasting them with Bik's. In Chapter 5, we will enumerate the major phases of our design and illustrate each by walking an example program through the compilation process. The remainder of the thesis will flush out the details of each of these phases.

Chapter 2

Previous work in sparse compilation

Several approaches have been suggested for developing sparse compilers. One approach is to allow the user to write a dense specification of an algorithm and then to performance extensive run-time scheduling and optimization in order to exploit the sparsity of the run-time data ([57]). A different approach is to have the user write a dense specification and then provide annotations that inform the compiler where important dense-to-sparse or reindexing operations are to occur ([102]).

While these approaches hold promise for certain classes of computation, we are interested in pushing the concept of sparse compilation to its extreme limit. In that vein, we are interested in studying techniques that would be required to build a general purpose sparse compiler that requires as little direction from the user as necessary. Most of the previous work that has been done in this direction has been done by Bik ([19]). In this chapter, we give an overview of his compiler.

2.1 Bik's design goals

Bik's is a restructuring sparse compiler. This means that it reads the user's dense specification expressed as a conventional program, performs certain transformations, and then produces the transformed program that constitutes the sparse implementation. Bik's compiler reads a dense FORTRAN program and produces a corresponding sparse program.

The design goals of Bik's sparse compiler are to

- Accept "real" FORTRAN as input.
- Allow annotations in the program text to indicate the non-zero structure of sparse matrices.
- Select data structures appropriate for the sparse matrices.
- Transform the dense program to exploit the sparsity of the computation and these sparse data structures.

Using the terminology from Chapter 1, his compiler makes decisions and transformations for stages 2 and 3 of the process of exploiting sparsity.

Bik's sparse compiler will accept FORTRAN code from any of the four classes shown in Figure 1.1. In fact, his aim is to accept the dense BLAS and LAPACK library routines for these computations directly. The only changes that the user has to make to the program text is to add annotations to the declaration of each sparse matrix.

The declaration of a sparse array, A, must be annotated with a set of *property summaries*, $\mathcal{P}_A = \{\langle P, \mathbf{p}, p\rangle\}$. An individual property summary, $\langle P, \mathbf{p}, p\rangle$, describes the sparsity of a single region of that matrix, where,

- P specifies the region of A being summaries. In Bik's design the region must be described using a polygon.
- p is the sparseness[1] of the region, and is one of "dense", "sparse", or "zero".
- $\mathbf{p}$ is the direction of storage. This direction indicates whether the region is to be stored in a row, column, or diagonal manner.

Property summaries can be specified as annotations by the user, or they may be computed automatically for a sparsity analysis tool. In order for the property summaries to be determined automatically, the user must specify a file that contains the non-zero entries of the sparse matrix. The sparsity analyzer will read and analyze the non-zero entries and produce the appropriate property summaries. In Bik's compiler, sparsity analysis is done during the compilation process, but it could be done separately as well.

[1]This is our term, not his.

Bik's sparse compiler will transform a dense implementation into a sparse specification in three phases,

1. Program Analysis
2. Data Structure Selection
3. Sparse Code Generation

We will explain each in turn.

2.2 Program analysis

During this phase, the user's program is read and certain information is computed for use in later phases. The two primary sets of information computed are the sparsity guard and access summaries.

2.2.1 Sparsity guard

When transforming a dense specification into a sparse implementation, a sparse compiler must identity cases when computation can be avoided because of sparsity. The *sparsity guard* of a loop body expresses the conditions under which the body must be executed. When the sparsity guard is not true, the computation of the loop body has been proven to be useless and can be skipped. For instance, if A is a sparse matrix, the statement,

$$Y[i] := Y[i] + A[i,j] * X[j];$$

only has to be executed when $A[i,j] \neq 0$.

Instead of using the guard shown above exactly, Bik instead uses a weaker version of it. This is safe because, if the weaker sparsity guard is true when the original sparsity guard is not, then the final sparse implementation simply performs some useless computation. The weaker sparsity guard is constructed based on the following observations:

- If a entry is not stored in A, then its value is 0. The user must insure that this requirement is satisfied when placing data into a sparse matrix format. If this requirement is not satisfied, then the sparse compiler will not be able to generate code correctly.[2]

- If a value is stored in A, then it is very, very likely not to be 0. This is assumed to be true, but if it is not, then the generated code will still run correctly, albeit slowly.

The weaker form of the sparsity guard is obtained by taking the original sparsity guard and replacing all occurrences of $M[\alpha, \beta] \neq 0$ with $BVAL(M[\alpha, \beta])$, where $BVAL(M[\alpha, \beta])$ is true when $M[\alpha, \beta]$ is stored in the sparse matrix M.[3] The advantage of this weaker sparsity guard over the original sparsity guard is that it is almost certainly cheaper to evaluate. This is because the weaker test simply checks for the existence of an entry, while the original test requires checking for its existence and then testing that its value is non-zero.

But does this substitution, in fact, result in a weaker guard? Certainly, if all of the terms of the original guard are of the form $M[\alpha, \beta] \neq 0$, then, since $A[i, j] \neq 0 \Rightarrow BVAL(A[i, j])$, the guard is weaker. But what if the guard contains terms of the form $M[\alpha, \beta] = 0$? As we will see in Chapter 8, it does not, so the new guard is weaker.

Because the original form of the sparsity guard is not used, when we refer to a "sparsity guard", we will mean this weaker form of the sparsity guard.

In order to show how the sparsity guard can be computed, we will start by examining the simplest, non-trivial loop body, a single assignment statement. Consider the example statement,

$A[i] := B[j];$

The sparsity guard for this statement is given by,

$$BVAL(A[i]) \vee BVAL(B[j]).$$

This can be seen by examining the four possible situations that might occur.

---

[2]One can imagine computations in which an entry not being stored meant that it has some other value (e.g., 1 or ∞). Bik does not consider these sorts of computations, and neither do we.

[3]$BVAL(\ldots)$ is our notation, not his.

- $BVAL(A[i]) \wedge BVAL(B[j])$. The value if $B[j]$ must be written to $A[i]$.
- $\neg BVAL(A[i]) \wedge BVAL(B[j])$. The entry for $A[i]$ does not exist. It must be created and initialized with $B[j]$.
- $BVAL(A[i]) \wedge \neg BVAL(B[j])$. The value of $B[j]$ is 0. The entry for $A[i]$ may be deleted.
- $\neg BVAL(A[i]) \wedge \neg BVAL(B[j])$. Both entries are missing, so both values are implicitly 0. The statement does not need to be executed in this case.

The disjunction of the first three conditions simplifies to the sparsity guard given above.

In the general case, the statement

$$var := rhs;$$

must be executed when either there is storage allocated for var or the rhs expression is non-zero. If we define $ALC(var)$ as the boolean expression indicating whether or not storage is allocated for var, and $NZ(rhs)$ as the boolean expression indicating whether or not the rhs is non-zero, then we can define $SP(\dots)$, the sparsity guard for this statement, as,

$$SP(var := rhs) = ALC(var) \vee NZ(rhs);$$

where $NZ(rhs)$ and $ALC(var)$ are defined as attribute grammars below.

$NZ(e)$ will traverse an expression, e, and returns a boolean expression that indicates when it might be non-zero. This is done by examining the arithmetic operators of e and using their algebraic properties to determine the effects of zeros on the results of the expression. Here is the definition of NZ, in which $\#t$ is the boolean value "true" and $\#f$ is the boolean value "false":

$$\begin{aligned} NZ(e1+e2) &= NZ(e1) \vee NZ(e2) \\ NZ(e1-e2) &= NZ(e1) \vee NZ(e2) \\ NZ(e1*e2) &= NZ(e1) \wedge NZ(e2) \\ NZ(e1/e2) &= NZ(e1) \\ NZ(e1^{e2}) &= NZ(e1) \\ NZ(var) &= ALC(var) \\ NZ(0) &= \#f \\ NZ(const) &= \#t \end{aligned}$$

$ALC(var)$ traverse a variable reference, var, and returns a boolean expression that indicates when there is storage allocated for var. If var is a dense array reference or scalar variable, then $ALC(var)$ is true. In these cases, there will always be storage for the reference. If var is a sparse array reference, then the appropriate $BVAL$ expression is returned.

$$\begin{aligned} ALC(v[k]) &= \#t, \text{ if } v \text{ is dense.} \\ ALC(scalar\_var) &= \#t \\ ALC(v[k]) &= BVAL(v[k]), \text{ if } v \text{ is sparse.} \end{aligned}$$

This completes the sparsity guard computation for a simple assignment statement. The attribute grammars presented here are variations on those developed by Bik and presented in [19], along with extended grammars to handle multiple statements, conditional statements, loops, and so on.

2.2.2 Access summaries

The second set of information collected during program analysis is a summary of every sparse matrix reference. The *access summary* for a reference to a sparse matrix, A, is a tuple $\mathcal{X}_A = \langle X, \mathbf{x} \rangle$, where X is the region of A accessed by the reference. $\mathbf{x}$ is the direction in which entries of A are touched as the index of the innermost loop that appears in A's access function changes. For instance, if a reference $A[i+j, j]$ appears in an $\langle i, j, k \rangle$, loop nest, the innermost loop index that appears in the access function is j, and the access direction is $(1,1)^T$. These directions are normalized for ease of use. This canonical representation of access is called the *normalized access direction*. See [19] for further details.

Example 2.1 In the following loop nest, taken from [19],

for $i_1 := 1$ to 4 do
    for $i_2 := i_1$ to 4 do
        $\ldots\ A[9 - 2 * i_2, i_1]\ \ldots$
    end do
end do

the i_2 loop is the innermost loop whose index appears in the array reference. The access direction of i_2 within A is $(2, 0)^T$, which normalized is $(1, 0)^T$.

Example 2.2 In the following loop nest, also taken from [19],

for $i_1 := 1$ to 4 do
    for $i_2 := 1$ to 50 do
        $\ldots\ A[10, 2 * i_1]\ \ldots$
    end do
end do

the i_1 loop is the innermost loop whose index appears in the array reference. The access direction of i_1 within A is $(0, 3)^T$, which normalized is $(0, 1)^T$.

2.3 Transformations

Recall that the user provides the compiler with property summaries, a description of the sparsity of each sparse matrix, and the compiler computes access summaries, a description of how each sparse matrix is accessed. In the transformation phase, the sparse compiler must select storage for each sparse matrix and transform the computation in order to access the selected storage as efficiently as possible. In order to accomplish this, the sparse compiler will select a set of "representatives" for each sparse matrix. Each representative will correspond to a different region of the sparse matrix, and may have its own storage format. The access summary information can be used to identify which representative is touched by each array access, and loop transformations can be performed in order to improve that access.

2.3.1 Properties of the representatives

More formally, the *representatives*, Σ_A, of a sparse matrix are a set of subsets of the indices of A that must satisfy the following properties,

1. $\cup_i S_i \in \Sigma_A = A$. The representatives cover A exactly.

2. $\forall S', S'' \in \Sigma_A, S' \neq S'' \Rightarrow S' \cap S'' = \phi$. The representatives are disjoint.

The first property ensures that the representatives cover the indices of A, and the second ensures that none of the representatives overlap. These two properties allow the sparse compiler to assign storage to the representatives and then to use the representatives in place of A.

3. $\forall \langle X, \mathbf{x} \rangle \in \mathcal{X}, \exists S \in \Sigma_A, X \subseteq S$. Each reference to A can be associated with a single representative.

This property allows the sparse compiler to generate code that uses the representatives in place of A, *without* having to determine at run-time which representative is being touched by a reference. This requirement is not strictly necessary; it reflects a design decision made by Bik.

In addition to these three requirements, there is a property that it is desirable, but not necessary, for representatives to have.

4. $\forall S \in \Sigma_A, \forall \langle P, \mathbf{p}, p \rangle \in \mathcal{P}_A, S \cap P \neq \phi \Rightarrow S \subseteq P$. The representatives correspond to fragments of the property summaries.

The original property summaries contained, p, an indication of which a region of the sparse matrix is entirely dense and entirely zero. If the representatives are subsets of these regions, then these sparsity properties are true of the representatives as well. If the representatives are not subsets, then these properties do not necessarily transfer, and the compiler must conservatively assume that the representatives are sparse. Thus, it is desirable for the representatives to be subsets of the original property summaries.

2.3.2 Computing the representatives

The representatives can be found an iterative computation. The set Σ_A is initialized with all P's, where $\langle P, \mathbf{p}, p \rangle \in \mathcal{P}_A$. By construction, this set satisfies properties 1 and 2 (and 4), but may not satisfy property 3. So, while there exists $\langle X, \mathbf{x} \rangle \in \mathcal{X}$, such that there is no $S \in \Sigma_A$ such that $X \subseteq S$, two sorts of refinements are heuristically made.

Coarsen Σ_A. The first refinement is to coarsen Σ_A. That is, if a set of regions $\{S_1, \ldots, S_n\} \subseteq \Sigma_A$ can be found such that $X \subseteq \cup_i S_i$, then all of these S_i's are removed from Σ_A and the single region $\cup_i S_i$ is added. This represents a tradeoff between property 4 and requirement 3.

Iteration splitting. The alternative to changing Σ_A is to change $\langle X, \mathbf{x} \rangle$. Recall that this access summary corresponds to a single array reference in the program. Instead of changing Σ_A, perhaps we can change the program so that this single reference, $\langle X, \mathbf{x} \rangle$, is split into several equivalent references $\langle X_i, \mathbf{x} \rangle$, each of which satisfies requirement 3.

Example 2.3 Consider the following loop nest taken from [19],

```
for i := 1 to 100 do
    for j := 1 to i − 1 do
        A1[i, j] := C[i, j];
    end do
    B(i) := A2(i, i);
    for j := i to 100 do
        A3[i, j] := D[i, j];
    end do
end do
```

where A_1, A_2, and A_3 denote three distinct references to the same sparse matrix A. Suppose that Σ_A consists of three representatives for the lower triangle, the main diagonal, the upper triangle of A. In this case, the A_3 reference violates requirement 3, since it touches both the main diagonal and the upper triangle.

On the one hand, the sparse compiler might choose to resolve this conflict by combining the main diagonal and upper triangle representatives into a single representative. This coarsening would result in less efficient storage if, for instance, annotations had indicated that the main diagonal was dense and the upper triangle was zero. It also means that the compiler has to generate a search for the A_2 reference.

On the other hand, the sparse compiler might choose to split the first iteration from the loop in which the A_3 appears:

```
for i := 1 to 100 do
```

```
        for j := 1 to i − 1 do
            A_1[i, j] := C[i, j];
        end do
        B(i) := A_2(i, i);
        A_3a[i, i] := D[i, i];
        for j := i + 1 to 100 do
            A_3b[i, j] := D[i, j];
        end do
    end do
```

By doing so, the A_3 reference is split into two references, A_{3a} and A_{3b}. Neither of these new references violates requirement 3, so the conflict has been resolved.

Since splitting the iterations can resolve conflicts without coarsening the representatives, why not have the sparse compiler always choose this refinement? One reason is that dependences in the code may not allow such a code transformation to be made. Another is that each splitting increases the size and overhead of the code; after some point, it is no longer profitable to use this refinement.

2.3.3 Data structure selection for the representatives

After Σ_A has been chosen, storage formats must be selected for each of the representatives. In order to do this, the sparse compiler assigns a sparseness and a direction of storage to each representative, $S \in \Sigma_A$. If the representative S is a subset of one of the property summaries, $\langle P, \mathbf{p}, p \rangle$, then the sparseness and direction of storage of the representative can be taken from the property summary and will be p and $\mathbf{p}$ respectively. Otherwise, the representative is conservatively assigned the "sparse" sparseness, and a direction is chosen that will tend to store the region in a few, very long vectors. Further details can be found in [19].

2.3.4 Loop transformations

Once the direction of storage has been selected for each representative, the sparse compiler attempts to transform the loop nests so that access will occur along this direction of access.[4] Bik uses the usual integer linear framework for reasoning about these loop transformations. If the reader is unfamiliar with this framework, suitable introductions can be found in [12] and [121].

Let $(s_1, s_2)^T$ be the prefered direction of access for a single sparse matrix reference,

$$F(i_1, \ldots, i_n) = \mathbf{W} \begin{pmatrix} i_1 \\ \vdots \\ i_n \end{pmatrix} + \mathbf{v} = \mathbf{W}\mathbf{i} + \mathbf{v}$$

In order to obtain the most efficient access of the sparse matrix, the sparse compiler will looks for a loop transformation, $\mathbf{U}$, that will produce a new iteration space, $\mathbf{i}' = \mathbf{U}\mathbf{i}$, in which the sparse matrix reference $F(\mathbf{U}^{-1}\mathbf{i}')$ will be along the prefered direction of access. That is, accesses made by the innermost loop of the new loop nest will lie on a vector with the direction $(s_1, s_2)^T$. When this is the case, then the following condition will hold,

$$0 = \begin{pmatrix} s_2 \\ -s_1 \end{pmatrix}^T \mathbf{W} \underbrace{\mathbf{U}^{-}1 \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}}_{\alpha} = \begin{pmatrix} s_2 \\ -s_1 \end{pmatrix}^T \mathbf{W}\alpha$$

The term, $\mathbf{W}\alpha$, gives the direction in which elements of the array are accessed as the new inner loop index is incremented. The vector, $(s_2, -s_1)^T$, is orthogonal to the prefered direction, $(s_1, s_2)^T$, so we want the dot-product of this vector and the access direction to be 0. This give us the condition necessary to align a single reference with its prefered direction of access.

For multiple references, we want to find a loop transformation, $\mathbf{U}$, that will satisfy this condition for each of the references. Given, c sparse array references, if we define $\mathbf{S}$ as the system,

[4] Actually, Bik performs these transformations *before* selecting the representatives. It not clear why this is preferable to performing them afterward.

$$\mathbf{S} = \begin{pmatrix} s_{2_1} & -s_{1_1} & & & \\ & & \ddots & & \\ & & & s_{2_c} & -s_{1_c} \end{pmatrix} \begin{pmatrix} W_1 \\ \vdots \\ W_c \end{pmatrix}$$

then we would like to find a vector, α, such that $\mathbf{S}\alpha = 0$. Any vector in the null space of $\mathbf{S}$ can be used. Once α is found, it can be used as the last column of $\mathbf{U}^{-1}$. The sparse compiler has ensured that each sparse array reference is associated with only one representative, so each reference need only be transformed to satisfy one direction constraint. However, there may not be a single loop transformation that will satisfy the direction constraints arising from all of references in a loop. Also, dependencies may not allow certain direction constraints to be satisfied. For these reasons, the sparse compiler must heuristically select the loop transformation that will mostly maximize the performance of the final code. Bik provides a completion procedure, which will "grow" this single column into a complete unimodular $\mathbf{U}$ that will satisfy as many of the constraints as possible without violating any dependencies that might be present in the original loop nest.

2.4 Code generation

After storage formats has been assigned to each representative, and after the loop transformations have been performed, the sparse compiler transforms the dense specification into the sparse implementation.

2.4.1 Generating storage

First, data storage must be allocated for the representatives of each sparse matrix.

- If a representative is marked "dense", then a small array will be generated for its storage. The direction of storage will be used to determine whether the storage will be row-major, column-major, or oriented in some other direction.
- If a representative is marked "sparse", then a set of sparse vectors will be used for its storage. That is, the direction of storage determines

a set of vectors in which the entries of the region are stored. These vectors are sparse because only the non-zero entries should be stored.

- If a representatives is marked "zero" then there is no need to store it at all.

Next, local transformations are made to the code in order to obtain efficient access of the representatives. In order to illustrate the transformations, we will use an example: suppose that the following loop nest appears after selecting the representatives and performing the loop transformations,

```
for i := 1 to n do
    for j := 1 to n do
        sum := sum + A[i, j] * B[i, j];
    end do
end do
```

where A and B are sparse matrices. The sparsity guard for the body of this loop is $BVAL(A[i,j]) \wedge BVAL(B[i,j])$. Suppose that a single representative is chosen for each of A and B, and that, in each case, the representative is given a row-oriented direction and is marked "sparse".

2.4.2 Generating Searches

At this point, the sparse compiler has determined the sparsity guard for the loop and the storage format for each representative touched by the loop. The sparse compiler could trivially generate code that searches the sparse matrices for the appropriate entries and explicitly tested the sparsity guard before executing the loop body.

```
for i := 1 to n do
    for j := 1 to n do
        (flag_A, ref_A) := search(A[i, j]);
        (flag_B, ref_B) := search(B[i, j]);
        if flag_A ∧ flag_B then
            sum := sum + ref_A * ref_B;
        end if
    end do
end do
```

However, this code is horrendously inefficient since an expensive search is being performed for each iteration of the original loop nest. Depending upon how the search is performed, the resulting implementation could take $O(n^2 \log n)$ time. While searches certainly have to be used in the general case, there are many common situations when the sparse compiler can generate more efficient code.

2.4.3 Guard Encapsulation

One technique for eliminating search is *guard encapsulation*. In this case, a single term of the sparsity guard is "folded" into the inner loop, so that only those iterations that satisfy the term will be accessed.

for $k := lb_k$ to ub_k do
    if $\ldots BVAL(A[F(\ldots, k)]) \ldots$ then
        body;
    end if
end do

$\Downarrow$

for $\{\langle k, ref\rangle | BVAL(A[F(\ldots, k)])\}$ do
    if $lb_k \leq k \leq ub_k$ then
        body;
    end if
end do

In order to perform guard encapsulation on a loop and a term, several properties must be true,

- The term being encapsulated must dominate the sparsity guard. A term ψ *dominates* the predicate ϕ when $\phi \Rightarrow \psi$ is true. By folding ψ into the loop, we ensure that only the computations for which ψ is true will be performed. This is safe, since $\neg\psi \Rightarrow \neg\phi$.

- The loop must be the innermost loop whose index appears in the access function of the term.

- If $\langle X, \mathbf{x} \rangle$ is the access summary of the reference associated with the term ψ and $\langle \mathbf{S}, \mathbf{s}, s \rangle$ is the representative associated with the reference, then $\mathbf{x} = \mathbf{s}$ must hold.[5] That is, the direction in which the loop will access the elements of the sparse matrix must be in the same direction in which they are stored. This means that all of the elements touched by the loop lie on the same vector of storage within the representative.
- The iterations of the loop can safely be performed in any order. This is because the non-zeros of the representative may be not sorted, and so there is no guarantee that they will be enumerated in any particular order.

For a given loop, there may be several terms that are candidates for guard encapsulation. In this case, the sparse compiler must heuristically select one for this optimization.

Example 2.4 Both the A and B references in the running example are candidates for guard encapsulation. If the sparse compiler chooses to fold the A reference into the j loop, then the code after the transformation will be,

```
for i := 1 to n do
    for {⟨j, ref_A⟩|BVAL(A[i, j])} do
        sum := sum + ref_A * B[i, j];
    end do
end do
```

In this code, there is no test that j falls within the interval $[1, n]$, the bounds of the original loop nest. This test can be eliminated when it can be shown that the array bounds guarantee that all indices produced from the array lie within the bounds of the original loop nest.

2.4.4 Access pattern expansion

Bik's sparse compiler will only fold one term into a loop using guard encapsulation. This does not mean, however, that remaining sparse references

[5]Directions are normalized, so they can be easily be compare.

must be handled using search. Efficient access may be obtained for additional references by performing *access pattern expansion*. An access pattern, $A[F(\dots,k)]$, is a candidate for expansion when,

- It satisfies the conditions for guard encapsulation listed above, and
- It can be shown that any other reference to A that occurs in the loop does not touch any entry in the vector $A[F(\dots,:)]$

For each spares matrix reference that is a candidate for this optimization, the non-zero entries in $A[F(\dots,:)]$, the vector that the loop will access, are *scattered* to a dense vector, V, before the loop is executed. The reason for scattering to a dense vector is that it provides fast access to each element. After the loop is executed, the non-zero entries are *gathered* from the dense vector back into the sparse matrix. This is shown in the following code:

for $k := lb_k$ to ub_k do
    $\dots\ A[F(\dots,k)]\ \dots;$
end do

$\Downarrow$

-- *Scatter* $A[F(\dots,:)]$ *to* V.
for $\{\langle k, ref\rangle | BVAL(A[F(\dots,k)])\}$ do
    $BVAL(V[k]) := \#t;$
    $V[k] := ref;$
end do
-- *Use* $V[k]$ *instead of* $A[F(\dots,k)]$.
for $k := lb_k$ to ub_k do
    $\dots\ V[k]\ \dots;$
end do
-- *Delete existing* $A[F(\dots,:)]$.
for $\{\langle k, ref\rangle | BVAL(A[F(\dots,k)])\}$ do
    $BVAL(A[F(\dots,k)]) := \#f;$
end do
-- *Gather* V *to* $A[F(\dots,:)]$.
for $\{k | BVAL(V[k])\}$ do
    $BVAL(A[F(\dots,k)]) := \#t;$
    $A[F(\dots,k)] := V[k];$

```
end do
```

Access pattern expansion exactly corresponds to the sparse accumulator load and store operations of Sparse MATLAB ([58]), and are similar to the scatter and gather operations found on some vector machines ([65]).

In our running example, the only sparse reference that is a candidate for expansion is $B[i,j]$. The code after this transformation will be,

-- *Scatter* $B[i,:]$ *to* V.
for $\{\langle j, ref_B\rangle | BVAL(B[i,j])\}$ do
    $BVAL(V[j]) := \#t;$
    $V[j] := ref_B;$
end do
-- *Use* $V[j]$ *instead of* $B[i,j]$.
for $i := 1$ to n do
    for $\{\langle j, ref_A\rangle | BVAL(A[i,j])\}$ do
        $sum := sum + ref_A * V[j];$
    end do
end do

Notice that V is not gathered to B after the computation. This is not required since B is only read in the loop.

2.5 Summary

To summarize, Bik set the following goals for his compiler,

- The sparse compiler accepts dense specifications written in "real" FORTRAN.
- The user must specify the non-zero structure of the sparse matrices.
- The sparse compiler is responsible for choosing a storage format for each sparse matrix.

and the following are the high points of his design,

- A set of annotations is provided with which the user can specify the property summaries of the sparse matrices.

- Representatives will be stored either as a dense array or as a set of sparse vectors. Representatives that are marked “zero” are not stored at all.
- The program is transformed in a manner that tends to “align” it with certain directions of access that are obtained from the array accesses and the sparse array storage.
- The sparse compiler will attempt to reduce the overhead present of accessing the sparse matrices by performing guard encapsulation and access pattern expansion along these directions.

As will be discussed in Chapter 4, we see several problems with this design.

- It relies upon being able to assign prefered directions of access to sparse matrix storage formats in terms of its row and column indices. This is not possible except for a restricted class of sparse matrix formats.
- The compiler assumes responsibility for assigning storage formats to sparse matrices and does not provide a mechanism for the user to do so.
- The compiler's range of storage format choices is fixed and small. The user has no way of adding to these choices.

Chapter 3

Relational Databases

In this chapter, we introduce important concepts from the database literature. First, we introduce the concept of a relation and then show how queries can be expressed using the relational algebra. Next, we show how query optimization techniques can be used to discover efficient evaluation schedules for these queries. Finally, we will introduce an extended relational algebra in order to be able to express a wider range of queries.

This is not meant to be a general introduction to relational databases. Rather, it is meant to introduce the reader to concepts that are used in the rest of this thesis. More complete introductions to relational databases can be found in [120] and [105].

3.1 Relational databases

3.1.1 Relations

A *relation* is a collection of tuples of a particular arity. Each of the positions of these tuples correspond to a particular *field* of the relation. A relation can be thought of as a table, with each row corresponding to a tuple, and each column corresponding to a field.

Notation 3.1

When giving a relation name, R, we will often the *schema*, or list of field names, of R as well. This is done as, $R(f_1, f_2, \ldots)$. We will not list the fields of R if the are not important in the particular context.

The notation $R.f$ refers to a particular field, f, of the relation, R, and is used to differentiate this field from another field of the same name in a different relation.

Example 3.1 An example of such a relation, $Tagged$, is shown below

Tag #	Species	Location tagged
001	Lion	Serengeti
010	Caribou	Alaska
251	Bald Eagle	Yosemite Park
477	Bactarian Camel	Gobi Desert
672	Kangaroo	Australian Outback
999	Grad. Student	Chapter House

When referring to this relation, we might use $Tagged(Tag\#, Species, Location)$, and we would refer to the $Location$ field of this relation as $Tagged.Location$.

3.1.2 Classical Relational Algebra

There are two fundamental means of expressing queries to a relational database system. The first is the *relational calculus*. There are actually many different relational calculi; the one we will use in this thesis will be the usual first-order predicate calculus.

Example 3.2 A query in this calculus might look like the following,

$$\{\langle i, j\rangle | \exists v, w, P(i, j, v) \wedge Q(i, j, w) \wedge 0 \leq i\}$$

The second notation is the classical relational algebra, which is an algebra with the following operators,

Notation 3.2

$\sigma_P R$ – The *selection* operator produces a relation containing the tuples of R that satisfy the predicate P.

$\pi_{\langle f_1, f_2, \ldots \rangle} R$ – The *projection* operator performs a projection on the tuples of R. The results are obtained by removing all but the fields, $\langle f_1, f_2, \ldots \rangle$, from each tuple in R and removing any duplicate tuples. The short-hand, $\pi_R S$, will mean "project S onto the fields of R", and $\pi_{\neg f} R$ will mean "project R onto all fields except f".

$R_1 \times R_2$ – The result of the *cross product operator* is obtained by concatenating each tuple of R_1 with each tuple of R_2.

$R_1 \bowtie_{R_1.f \Theta R_2.g} R_2$ – This operator is called the Θ-*join*. This operator finds pairs of tuples r_1 from R_1 and r_2 from R_2 that satisfy the constraint $r_1.f \Theta r_2.f$, where Θ is a arithmetic comparison operator (i.e., $=$, $<$, $\leq$, and so on.). Each such r_1 and r_2 are concatenated to form a single tuple in the result. This operator can be defined algebraically as,

$$R_1 \bowtie_{R_1.f \Theta R_2.g} R_2 = \sigma_{R_1.f \Theta R_2.f}(R_1 \times R_2)$$

The *equi-join* operator refers to the common case when Θ is $=$.

$R_1 \bowtie R_2$ – This operator is called the *natural join*. This operator,

- finds all tuples r_1 from R_1 and r_2 from R_2, where are the fields that common to both R_1 and R_2 are equal in both r_1 and r_2, and
- for each such r_1 and r_2, adds a tuple to the result that is obtained by concatenating r_1 and r_2 and deleting the duplicate occurrences of the common fields.

This operator can be defined algebraically as,

$$R_1 \bowtie R_2 = \pi_{g_1, \ldots, g_n} \sigma_{R_1.f_1 = R_2.f_1 \wedge \ldots \wedge R_1.f_m = R_2.f_m}(R_1 \times R_2,)$$

where each g_i is a field that exists in either R_1 *or* R_2, and each f_j is a field that exists in R_1 *and* R_2. For clarity, we will often explicitly indicate the fields that are being joined as, $R_1 \bowtie_{g_1, \ldots, g_n} R_2$.

$R_1 \ltimes R_2$ – This operator is called the semi-join and is all tuples of R_1 that appear in $R_1 \bowtie R_2$. This operator can be defined algebraically as,

$$R_1 \ltimes R_2 = \pi_{R_1}(R_1 \bowtie R_2).$$

Example 3.3 One of the equivalent relational algebra expression for the relational calculus expression given above is,

$$\pi_{i,j}\sigma_{0 \leq i}(P(i,j,v) \bowtie_{i,j} Q(i,j,w))$$

The relational calculus and the relational algebra are equally expressive, and this result can be found in most any database textbook. Because of their equivalent expressiveness, the two will be used interchangeably in this thesis.

Notation 3.3 (SQL)

In a few places we will express a queries in the relational query language, SQL ([32], [33]). SQL queries have the the basic form,

SELECT field_1, field_2, ... , field_F
FROM rel_1, rel_2, ... , rel_R
WHERE pred

which is to be interpreted as,

1. From the relations, rel_1, rel_2, ... , and rel_R,
2. extract the tuples that satisfy the predicate pred,
3. delete all fields from the resulting tuples, except for field_1, field_2, ... , and field_F,
4. delete any duplicates and return the results.

This can be expressed concisely in the relational algebra as,

$$\pi_{\mathsf{field}_1,\ \mathsf{field}_2,\ \ldots\ ,\ \mathsf{field}_F}$$
$$\sigma_{\mathsf{pred}}$$
$$(\mathsf{rel}_1 \times \mathsf{rel}_2 \times \cdots \times \mathsf{rel}_R)$$

3.2 Query Optimization

A conventional Database Management System (DBMS) will accept a query in some form, schedule it for efficient evaluation and then evaluate the optimized query. A schematic for such a system is shown in Figure 3.1 (modified from a figure in [105]).

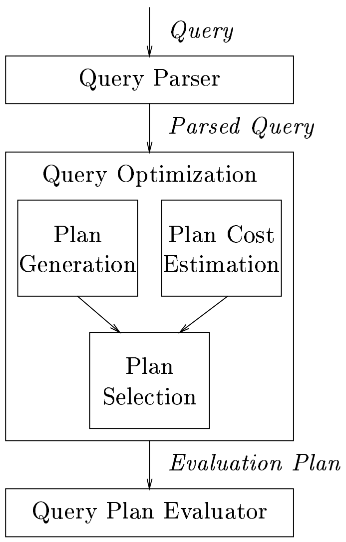


Figure 3.1: Schematic for a Database Management System

The core component of such a system is the *query optimizer*. The query optimizer takes a query and attempts to find a *plan*, or a sequence of concrete actions, for evaluating this query that maximized some utility function. Usually, this utility function is taken to be the inverse of the time required to evaluate the query. That is, the query optimizer will attempt to minimize the time required to evaluate the query. There are three basic tasks involved in query optimization

Plan Generation – This task is responsible for deterministically enumerating a finite set of plans for evaluating the original query.

Plan Cost Estimation – This task takes a single plan and estimates its running time using cost models of each of the operation appearing in the plan together with statistical information about the relations.

Plan Selection – This task is responsible for finding the plan from all of those produced by the Plan Generator that has the minimum running time, as specified by the Plan Cost Estimator.

We will discuss each of these tasks in turn.

3.2.1 The notations for plans

In order to enumerate all possible plans for evaluating a query, we need to specify how are the plans specified, and what are the steps for enumerating all of the plans that are equivalent to the original query.

There are many different notations used in the database literature to express plans (see [71] for examples) Here, we will use the one that is, perhaps the most intuitive: expression trees for relational algebra expressions. Consider the following SQL query (modified from an example in [70]) as the original query,

```
SELECT name, floor, balance
FROM emp, dept, acnt
WHERE emp.dno = dept.dno AND dept.ano = acnt.ano
    AND acnt.balance < 50000
```

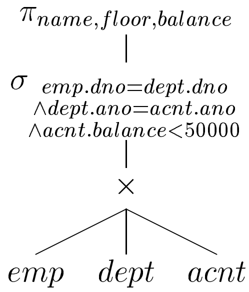
where the schemata for the relations that appear in this query are,

```
emp(name,age,sal,dno)
dept(dno,dname,floor,budget,mgr,ano)
acnt(ano,type,balance)
```

After parsing, this query can be represented as the relational algebra expression,

$$\pi_{name,floor,balance}\ \sigma_{\substack{emp.dno=dept.dno\\ \wedge dept.ano=acnt.ano\\ \wedge acnt.balance<50000}} (emp \times dept \times acnt)$$

which can be represented as the expression tree,



We will call such an expression tree a *high-level plan* because, if we think of data flowing through the edges of the tree from top to bottom, it represents a high-level description of the evaluation of the original query. If we assign particular implementations to each of the π, σ, and $\times$ operators, then we have a complete specification for the evaluation of the query. We call the relational expression tree with all operators assigned implementations the *low-level plan* for the query, or simply the plan.

3.2.2 Plan Generation

Having established the notation to be used, it remains to be seen how we can enumerate a finite set of equivalent plans from an initial expression tree. Here is how it is done:

1. A set of algebraic equivalences is used to transform the tree corresponding to the original query into an set of equivalent trees. This set of derived trees forms the candidate high-level plans.

2. The operators of each candidate high-level plan are assigned different feasible implementations in order to produce the final set of candidate low-level plans.

Enumerating the high-level plans

Suppose that the following two algebraic rules are used to transform the original expression tree,

$$\sigma_{P \wedge Q} R = \sigma_P \sigma_Q R$$
$$\sigma_{R.f=S.f}(R \times S) = R \bowtie_f S$$

The following expression trees are two of many that can be obtained by applying these transformations.

Even though these two expression trees define the same result as the original query, they are different high-level plans with potentially vastly different execution costs.

Using a set of algebraic rules does not guarantee that only a finite set of high-level plans can be generated. However, the set of candidate high-level plans can be made finite by placing additional restrictions on the expression trees considered. Some restrictions can be shown to exclude only suboptimal plans. An instance of this is the following:

- Do not consider plans containing $\times$'s when $\bowtie$ operators can be used ([70]).

Other restrictions are heuristic in nature. That is, applying them may remove the optimal plan from consideration, but this will usually not occur in practice. Here is an instances of a heuristic restriction:

- Only left deep trees of join operators are considered. Right deep and bushy trees are excluded ([112]).

Enumerating the low-level plans

Each high-level plan produced must be assigned implementations for each of its operators in order to produce a set of candidate low-level plans. Consider the $\bowtie$ operator: the database literature describes four basic implementations for computing the join $R_1 \bowtie_f R_2$ ([120], [105]),

Nested loop join. In this strategy, we compare all pairs of tuples from R_1 and R_2 in order to find those with equal f fields.

```
results := ϕ;
for t1 ∈ R1 do
    for t2 ∈ R2 do
        if t1.f = t2.f then
            results := results ∪ {⟨t1, t2⟩};
        end if
    end do
end do
```

Sort-Merge join. If R_1 and R_2 are sorted by f, then the results of the $\bowtie$ can be computed by simultaneously enumerating the tuples of R_1 and R_2. We do this by keeping a pointer into each relations. If the f field in the tuple pointed to by the R_1 pointer is equal to the f field in the tuple pointed to by the R_2 pointer, then we generate a tuple for the result relation, and advance both pointers. If the f field in the tuple pointed to by the R_1 pointer is less than than the f field in the tuple pointed to by the R_2 pointer, then we advance the R_1 pointer. Otherwise, we advance the R_2 pointer.

```
results := ϕ;
t1 := R1.first();
t2 := R2.first();
while valid(t1) ∧ valid(t2) do
    if t1.f = t2.f then
        results := results ∪ {⟨t1, t2⟩};
        t1 := R1.next();
        t2 := R2.next();
    else if t1.f < t2.f then
        t1 := R1.next();
```

```
        else
            t_2 := R_2.next();
        end if
    end do
```

Notice that this code is very similar to the two-finger dot-product shown in Figure 1.4.

Indexed join. If R_2 has an index on f, then we can enumerate the tuples of R_1 and use the index to find the appropriate tuples from R_2.

```
results := φ;
for t_1 ∈ R_1 do
    let T = σ_{t_1.f=f} R_2;
    for t_2 ∈ T do
        results := results ∪ {⟨t_1, t_2⟩};
    end do
end do
```

Hash join. If R_2 does not have an index on f, then one can be created on the fly.

```
results := φ;
initialize hashtbl;
for t_2 ∈ R_2 do
    add t_2 to hashtbl[t_2.f]
end do
for t_1 ∈ R_1 do
    for t_2 ∈ hashtbl[t_1.f] do
        results := results ∪ {⟨t_1, t_2⟩};
    end do
end do
deallocate hashtbl;
```

Notice that this code is very similar to the scatter version of dot-product shown in Figure 1.3.

A DBMS is designed with a wide array of implementations for each operator. However, this does not mean that every implementation for an operator is always feasible to use. For instance, the sort-merge join implementation, requires that the relations be sorted and cannot be used it they are not. In order to test for the feasibility of this implementation, the query optimizer needs to be able to determine whether the tuples of the relation are sorted by a particular field. This information is easily accessible from other components of the DBMS.

3.2.3 Plan Cost Estimation

In order to estimate the running time of individual low-level plans, the DBMS needs to provide the Plan Cost Estimator with certain information about each relation that is referenced in the query. For instance, the Plan Cost Estimator may need an estimate of the number of tuples in each relation and the distribution of keys within a particular index. This information is usually not completely accurate; not only it is expressive to compute exactly, but maintaining this information as the contents of the relations change has been found to be prohibitively expensive in practice ([112]). Often times, it is recomputed only on demand or during off-peak hours.

Accurate cost models of the performance of each of the operator implementations are also required in order to estimate the cost of an entire low-level plan. Such cost models are usually parameterized by the relation statistics discussed above. Simple cost models of many commonly used operator implementations can be found in [120] and [105].

3.2.4 Plan Selection

Given the set of candidate low-level plans, and given the method of estimating their cost, the final task of query optimization is that of determining the optimal plan. A brute force method of testing every plan could be used but is infeasible in practice. This is because the number of plans for a given query is, assuming a realistic set of algebraic transformations, is exponential in the size of the original query. In fact, it is shown in [69] that the query optimization problem is NP-complete, so no known polynomial time algorithm is known for solving the problem exactly. Since the query optimization problem is NP-complete, heuristic methods are used. A thorough summary of many planning heuristics presented in the database literature can be found in [70].

Here we will focus on what is perhaps the most popular, and what is certainly the earliest practical, method.

Selinger proposes in [112] a dynamic programming method for finding the optimal plan. In this method, only expression trees that satisfy the following two criteria are considered as candidate high-level plans,

- Only trees with join operators are considered.
- Only left deep join trees are considered.

The first criteria is not as restrictive as it appears: σ operators, for instance, can be pushed "into" the relation leaves for the purposes of query optimization. Also, π's can be incorporated into a join tree by placing annotations on the output of each join operator. The second criteria does in some cases exclude what would otherwise be the optimal plan, but it does not often do so in practice, and it dramatically reduces the number of high-level plans to be considered.

The dynamic programming method proceeds in stages, with as many stages as there are relations in the query. In the first stage, each relation referred in the query is considered and all possible implementations for enumerating its tuples are considered. The optimal implementation for each relation is selected and saved for the next stage; all others implementations are discarded. Thus, at the end of the first stage, each relation has associated with it a single plan for optimally enumerating its tuples.[1] During the second stage, all pairs of relations are combined using all feasible $\bowtie$ implementations. Again, the optimal plan for each distinct pair of relations is identified and retained for the next stage. At the end of the second stage, all pairs of relations have a single plan for optimally producing their join. This process is repeated until the last stage, after which a set of optimal plans for enumerating the join of all of these relations remains.

[1]This is not strictly true: a set of plans with varying costs and characteristics is retained. For instance, if two ways of enumerating a relation are available, one which enumerates the tuples in a sorted order and the other which does not, then both may be retained, even if one is cheaper than the other. This is so that subsequent stages can take advantage of the sorted enumeration if it is profitable to do so. This distinction is orthogonal to the main point of this discussion.

A diagram illustrating this process for the query $A \bowtie B \bowtie C$ is shown in Figure 3.2. Execution starts in stage 1 with the selection of the best plans for enumerating A, B, and C. In stage 2, the best plans form stage 1, as denoted by the arrows are used to construct the optimal plans two relation expressions. In stage 3, the best plans from stage 2 are combined with the best plans from stage 1 to produce the optimal plan for evaluating the entire query.

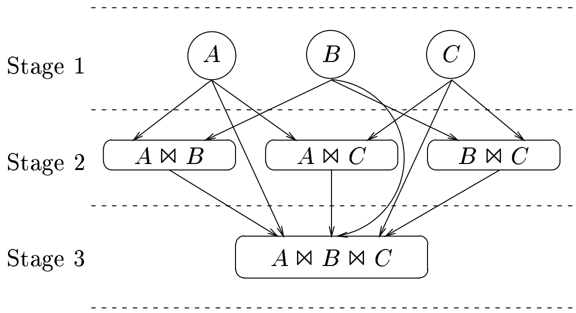


Figure 3.2: Plan Selection using Dynamic Programming

The attraction of the dynamic programming method is that, if certain monotone properties hold on the operator cost functions, then it is guaranteed to find the optimal evaluation plan. Furthermore, because most plans are removed from consideration well before the last stage, it will do so in practice in less than exponential time. Of course, for some queries, this method will still take exponential time.

3.3 The extended relational algebra

There are some circumstances where the operators of the classical relational algebra are not sufficient for expressing interesting queries. For instance, the $\bowtie$ operator discards tuples from either of its arguments when a match cannot be found in the other. For some queries, it is desirable to retain these unmatched tuples in the result.

Consider, for instance, the following query in English,

> Given the relations, $DataPt(t, value)$ and $Event(t, name)$, produce a time-line on which, for each time step, t, is listed all event $name$'s and data point $value$'s that occured at time t.

If we take the relations,

$DataPt$:

t	$value$
1	1.0
1	2.0
3	3.0

$Event$:

t	$name$
1	a
2	b
3	c

then the resulting time-line would be,

t	1	2	3
name	a	b	c
value	1.0 2.0		3.0

We would like to be able to express this query and its result in the relational algebra. However, this cannot be done without some modifications to its classical formulation.

3.3.1 Null values

The first question that must be addressed is: how can we represent the resulting time-line as a relation? Such a relation, $TimeLine$, might have fields t, $name$, and $value$, but what would the tuples of this relation be? Examining the time-line given above, we see that some t's have more than one data value, while others have none.

As with the classical relational algebra, we can represent multiple data values using multiple tuples. Thus, the following tuples would appear in $TimeLine$,

t	$name$	$value$
1	a	1.0
1	a	2.0
3	c	3.0

These tuples are simply the result of computing $Events \bowtie DataPts$.

How should we represent the fact that there is no data value for $t = 2$? The approach taken by the classical relational algebra is to not have any tuples for $t = 2$. In our case, this is not desirable, because this would mean that b, the event name of time step 2 would not appear anywhere in our $TimeLine$ relation. Instead, we will require that a tuple appear in $TimeLine$, with $t = 2$, $name = b$, and the $value$ field undefined. In other words, if ω is our "undefined value", then the tuple

$$\langle t : 2, name : b, value : \omega \rangle$$

will appear in the final result of $TimeLine$,

t	$name$	$value$
1	a	1.0
1	a	2.0
2	b	ω
3	c	3.0

This "undefined value" is referred to as the *null value* and it is used to denote the absence of a meaningful value in a particular field of a tuple.

3.3.2 New operators

Now that we see how the result of our query might be represented as a relation, the next question to answer is this: what operators must be added to the relational algebra in order to allow our English query to be written as a relational algebra expression? The operators that we wish to add are variants of the *outer join*.[2] We start by defining a more primitive operator from which we can derive these outer join variants.

Notation 3.4 (Anti-Join)

The *anti-join* of two relations, denoted, $R_1 \triangleright R_2$, is all tuples of R_1 that do not appear in $R_1 \bowtie R_2$, padded with ω's for the fields of R_2. For instance,

$$Events \triangleright DataPtrs = \{\langle t : 2, name : a, value : \omega \rangle\}$$

The anti-join operator can be defined algebraically as,

$$R_1 \triangleright R_2 = \underbrace{\underbrace{(R_1 - \underbrace{(R_1 \ltimes R_2)}_{(1)})}_{(2)} \times \underbrace{\Omega}_{(3)}}_{(4)}$$

Notes:

(1) This expression produces the tuples of R_1 that appear in the result of $R_1 \bowtie R_2$.

(2) This expression produces the tuples of R_1 that do *not* appear in the result of $R_1 \bowtie R_2$.

(3) Ω denotes a relation, whose fields the fields of R_2 that do not appear in R_1. Ω contains a single tuple whose fields are all set to ω.

(4) The result of the anti-join can be obtained by taking all of the tuples of R_1 that do not appear in $R_1 \bowtie R_2$ and crossing them with Ω. This has the effect of padding each of this tuples with ω's for each of the fields of R_2.

---

[2]The variants of the $\bowtie$ operator defined above are sometimes referred to as *inner joins*

Notation 3.5 (Outer join operators)
Using the $\triangleright$ operator, we can define several variants of the outer join operator,

- The result of $R_1 \leftrightarrow R_2$, called the *outer join*, contains,
  - the tuples of $R_1 \bowtie R_2$,
  - the tuples of R_1 not in $R_1 \bowtie R_2$, padded with ω's.
  - the tuples of R_2 not in $R_1 \bowtie R_2$, padded with ω's.

  This operator can be defined algebraically as,

$$R_1 \leftrightarrow R_2 = (R_1 \bowtie R_2) \cup (R_1 \triangleright R_2) \cup (R_2 \triangleright R_1)$$

- The result of $R_1 \rightarrow R_2$, called the *left outer join*, contains,
  - the tuples of $R_1 \bowtie R_2$,
  - the tuples of R_1 not in $R_1 \bowtie R_2$, padded with ω's.

  This operator can be defined algebraically as,

$$R_1 \rightarrow R_2 = (R_1 \bowtie R_2) \cup (R_1 \triangleright R_2)$$

Theorem 3.1

$$R_1 \leftrightarrow R_2 = (R_1 \rightarrow R_2) \cup (R_2 \rightarrow R_1)$$

Proof

$$\begin{aligned} R_1 \leftrightarrow R_2 \;& (R_1 \bowtie R_2) \cup (R_1 \triangleright R_2) \cup (R_2 \triangleright R_1) \\ & (R_1 \bowtie R_2) \cup (R_1 \triangleright R_2) \cup (R_2 \bowtie R_1) \cup (R_2 \triangleright R_1) \\ & (R_1 \rightarrow R_2) \cup (R_2 \rightarrow R_1) \end{aligned}$$

The classical relational algebra, together with null values and the outer join operators is referred to as the *extended relational algebra*. The English query given above can be written in this extended relational algebra as,

$$TimeLine = Events \leftrightarrow DataPts$$

There are many other interesting and important properties of null values and outer joins that we will not discuss here. More detailed discussions of the extended relational algebra can be found in [37], [44], and [45].

3.4 Summary

A relational database, or *relation*, is simply a table whose columns are fields and whose rows are tuples. The *relational algebra* is used for expressing queries about relations. The most important operator of this algebra is the *join* operator, of which there are many different variants:

Equi- vs. Θ: The Θ-join is used to relate the two join fields by a general comparison constraint. The equi-join restricts this constraint to being a simple equality.

Natural vs. not natural: A natural join operator removes multiple occurrences of the join field from the resulting tuples. Natural joins are generally only defined for equi-join operators.

Inner vs. Outer: An inner join is obtained by finding the tuples from both relations that satisfy the join constraint. An outer join contains these tuples, as well as any tuples that appeared in only one relation, padded with ω's. Only the natural versions of the outer join operators are usually defined.

We have discussed the standard methods for scheduling relational algebra queries for efficient evaluation.

Chapter 4

Our approach

While Bik's sparse compiler will provide general sparse implementations that deliver good performance in general, we are interested in obtaining the best possible performance for certain classes of applications. In particular, we are interested in obtaining peak performance for iterative solvers.

In this chapter, we will describe how our goals and parameters differ from Bik's. We will then enumerate the major points of our design and discuss how our approach differs from Bik's design. In Chapter 5, we will give an extended example to illustrate this approach.

4.1 Goals

As we mentioned earlier, the focus of our work in sparse compilation has been to obtain peak performance for iterative methods. The three most expensive operations performed in these computations are,

Matrix Vector Multiplication (MVM): $y = A * x$, where A is sparse and x and y are dense.

Triangular Solve: $x = L^{-1}b$, where L is sparse and lower triangular, or $x = U^{-1}b$, where U is sparse and upper triangular.

Computing the preconditioner: There are many ways that this can be done. Some popular preconditioners involve computing the incomplete factorization of a matrix. That is, either $\hat{L}\hat{L}^{-1} = \hat{A}$ or $\hat{L}\hat{U} = \hat{A}$.

In terms of the classification shown in Figure 1.1, these operations fall into Classes I, III, and IV, respectively.

At the end of Chapter 2, we suggested three problems with the design of Bik's compiler. Here we will discuss how each of these problems arise in the context of iterative solvers. Addressing each of these problems will be the goals of our compiler design.

Obtaining prefered direction vectors. One of the core assumptions of Bik's compiler design is that the efficient means of accessing sparse matrix storage formats can be described using prefered directions, which are vectors in an integer space. The vector is used to formulate a linear system in order to obtain loop transformation to ensure that direction of access.

However, many storage formats that are used in practice cannot be characterized in this manner. For instance, of the storage formats listed in Tables 1.1 and 1.2, only the Diagonal Skyline and CRS formats can be assigned meaningful prefered direction vectors. As we will see in Section 7.3, where these formats are all discussed in detail, the other storage formats provide efficient access to their entries, but not along either the rows, columns, or diagonals of the matrix. So, in order to produce efficient sparse iterative solvers for the storage formats that are used in practice, it is necessary to develop a transformation framework that is not based upon the usual integer linear model.

User selected storage formats. The results shown in Tables 1.1 and 1.2 demonstrate that a storage format must be carefully chosen to match the sparsity pattern, architecture, and algorithm in order to obtain maximum performance for these sorts of operations required by an iterative solver. Bik provides a heuristic method which his sparse compiler uses to assign storage formats to sparse matrices, but we do not believe that such a heuristic will be able to choose storage formats nearly as well as a knowledgeable user. Furthermore, even if heuristic methods are appropriate, we wish to have them separated from the rest sparse compilation process. For these reason, we wish to have the user make these decision instead of the compiler.

User specified formats. It is also the case that there are storage formats not shown in Tables 1.1 and 1.2 that will yield even better performance for certain sparsity patterns. If the user were to make a selection from such a fixed set of sparse matrix storage format choices, then the sparse compiler might be able to obtain good performance for all sparsity patterns, but it would not be able to obtain the best performance for certain patterns. Instead of having a fixed menu of formats from which the user must select, our sparse compiler should provide a mechanism with which the user can describe novel sparse matrix storage formats for the compiler to use.

Summary. To summarize, the following are the design high-level goals of the Bernoulli sparse compiler,

- This sparse compiler will accept dense specifications for part or all of an iterative method and will produce sparse implementation that are competitive in performance with those currently being written by hand.
- This sparse compiler will produce sparse implementations for the sparse matrix storage formats that are used in practice.
- This sparse compiler will allow the user to specify the format in which sparse matrices are to be stored.
- This sparse compiler will provide a mechanism for the user to describe novel storage formats.

4.2 Key aspects of the design

Since our goals and assumptions are different from Bik's, our design is necessarily different. Also, there are portions of Bik's design that we wish to improve. In this section, we will discuss some of the high-level aspects of our design.

4.2.1 Sparsity Annotations

Like Bik, our compiler will provide a set of annotations for the sparse matrix declarations. However, instead of specifying the non-zero structure of each sparse matrix, the annotations provided by our compiler allow the user to explicitly specify the storage format of the sparse matrix. We see several advantages with this approach. The first is that we believe that a knowledgeable user is able to select the most appropriate storage format with little effort. The second is that, since we want to give the user the option of specifying novel storage formats, the user must provide the name of the storage format in these circumstances.

However, explicit storage format directives do not preclude automatic methods of assigning storage formats. For instance, a separate tool can be built, perhaps using the methods described by Bik, that analyses the sparsity patterns of sparse matrices and selects appropriate storage formats. These storage format selections can then be conveyed to the sparse compiler using the explicit storage format directives annotations.

4.2.2 Black-box protocol

One of our design goals is to allow the user to add to the compiler's repertoire of sparse matrix storage formats. While there are many different ways that this could be done, we have chosen what we think to be the simplest and the most flexible.

As shown in Figure 4.1, the Bernoulli sparse compiler is implemented in a modular fashion. In particular, a set of modules is used to implement each of the sparse matrix storage formats. There is a single interface between these modules and the compiler that is well-defined and strictly enforced. Thus the compiler can treat each of these storage format modules as black-boxes. Hence, we call the interface between the compiler and the storage format modules the *black-box protocol*.

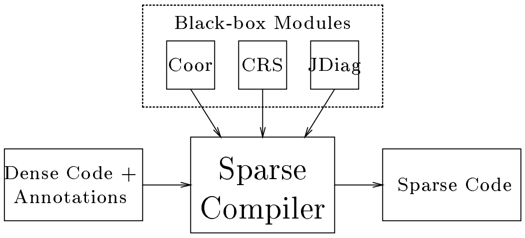


Figure 4.1: Organization of the Bernoulli Sparse Compiler

Because there is a single, well-defined interface between the storage format modules and the rest of the compiler, it is feasible for the user the provide their own module that implements the black-box protocol. Thus, to incorporate a novel storage format into the compiler, all the user has to do is provide the compiler with the appropriate module. The advantage of this approach over, say a complex set of annotations, is that, in principle, there are virtually no restrictions on the storage formats that the user can describe with the protocol.

The downside of this approach is that at the moment there is a fairly steep learning curve for the user who wants to add a new format to the compiler. We have made every effort to make the black-box protocol as simple and clean as possible. However, the user still understand the protocol and the compiler's internal program representation in order to implement these black-boxes modules. It might be possible to make the task easier by developing a Graphical User Interface (GUI) to the the protocol, but we have not done so at the current time.

We will discuss the information conveyed by the black-box protocol in Chapter 7. The actual interface that constitutes the protocol is presented in Appendix C.

4.2.3 Data-centric

Using the black-box protocol, the user is able to describe storage formats, like Coordinate, ITPACK, JDiag, and BlockSolve, that cannot be reasonably be assigned a prefered direction of access.

Consider Coordinate storage: in this case, each non-zero entry of the sparse matrix is placed in a "table" that simply records its row and column index and its value. The only operation that is provided for this format is to enumerate all of its non-zero entries. In particular, this format does not provide any efficient means to accessing all of the non-zero entries within a particular row or column.

Bik's approach to sparse compilation is iteration-centric. That is, his compiler takes a set of loop nests as input, performs certain transformations to these loops, and finally generates the resulting loops. The loop transformation technique implemented in Bik's sparse compiler and described in Section 2.3.4 require a prefered direction of access, $(s_1, s_2)^T$, which is used as the target of the loop transformations. What should be done with sparse matrices that are stored in Coordinate or other such format?

The obvious solution is to not consider references to such matrices when performing loop transformations. Unfortunately, the purpose of the loop transformations is to obtain efficient access to the sparse matrices. If these references are not considered, then there is no way to ensure efficient access to these matrices.

Instead of performing transformations on the original loops in order to gain efficient access to the sparse matrices, we take a different approach:

1. The compiler will extract as much information as we need from the original loops. After summarizing the loops we discard them.

2. The compiler will find efficient means of traversing the sparse matrices.

3. For each entry accessed by the sparse matrix traversal, the compiler will deduce the set of iterations that need to be performed.

4. The compiler generate the final code

We call this a *data-centric* approach to program transformation. Consider the following loop nest:

for $\mathbf{i} \in I$ do
  $\ldots\ A[F\mathbf{i}]\ \ldots$
end do

If we were to apply the data-centric approach in order to transform this code, we might get something like the following as our result,

for $\mathbf{a} \in A$ do
  for $\mathbf{i} \in F^{-1}\mathbf{a}$ do
    $\ldots\ A[\mathbf{a}]\ \ldots$
  end do
end do

If the original loop bounds and array access functions are affine, then existing systems such as Omega ([103]) and PIP ([54]) can be used to simplify the bounds of the transformed loop nest.

The advantage of the data-centric approach is that it can be used to obtain efficient access by sparse array storage formats whose efficient access cannot be easily expressed as directions within the iteration space. The data-centric approach has been applied to more traditional compiler problems, like blocking loop nests in order to increase cache locality ([80]).

4.2.4 Joins

If only one sparse array reference appeared in any loop nest in a dense specification, then it would sufficient to use the data-centric approach to generate sparse implementations. In particular, the sparse compiler could transform the loops so that they enumerate directly over the elements of the sparse array and use the original array reference to compute what iterations of the original loop nest must be performed for each element

When several sparse array references appears in a loop nest, in addition to finding efficient traversal orders for each sparse array, we need to find an efficient order for traversing *all* of the sparse arrays *simultaneously*. The following code for computing the dot-product of two sparse vectors, for instance, traverses the two sparse vectors in an efficient manner (i.e., by enumerating the non-zero elements of each vector directly), but the resulting implementation is very inefficient.

```
sum := 0;
for ii_X := 1 to nzs_X do
    for ii_Y := 1 to nzs_Y do
        if ind_X[ii_X] = ind_Y[ii_Y] then
            sum := sum + val_X[ii_X] * val_Y[ii_Y];
        end if
    end do
end do
```

Consider the sparse dot-product codes discussed in Section 1.6.3. There we illustrated two different ways of implementing this code more efficiently,

- By enumerating the non-zero entries of both vectors simultaneously (Figure 1.4), and

- By enumerating the non-zero entries of one vector and searching for the corresponding non-zero entry in the other (Figure 1.3).

In neither case is it sufficient to find an efficient traversal for each vector separately. In the first case, we need to find a way of traversing both vectors together, and in the second, we need to find a way of efficiently traversing the first vector and way of searching the second vector.

Fundamentally, what is occuring is that there are entries in each vector that must be "matched" by the sparse implementation. In the sparse dot-product example, the entries of X must be matches with the entries of Y on the i_X and i_Y indices, respectively. In MVM, the entries of Y must be matches with the row index of entries of A, and the column index of those entries of A must be matched with the index of entries of X.

We have seen a similar problem discussed in relational databases material: given a query in which the constraint $R_1.f = R_2.f$ appears, where R_1 and R_2 are two relations and f is a field that appears in both, the *join* operator can be used to find tuples from two relations that match in the field f. Discovering and scheduling these joins is the key to efficiently evaluating relational database queries. In particular, when given a relational query, a relational database system performs two tasks before evaluating the query,

- Joins that are explicitly or implicitly specified in the query are identified. In order to do this, the database system examines the constraints in the query and extracts constraints of the form $R_1.f = R_2.f$ in order to form joins.
- Efficient implementations of each join are selected depending upon such factors as the presence of indexing structures on the join fields. As we will discuss in Section 4.3.3, there are several different strategies that can be used to handle a variety of situations.

We have a very similar problem: in order to obtain efficient sparse implementations, we need to identify situations in which a constraint exists between several sparse matrices, and we have to select a strategy for efficiently enumerating the solutions of this constraint. We will borrow the idea of a "join" from the database literature, and adapt it to the problems of sparse compilation. Thus, our sparse compiler will perform two tasks analogous to those performed by query optimizers:

- The joins present in the sparse computation are identified, and
- Efficient implementations are chosen for each join, depending upon the efficient means of accessing the sparse matrix storage formats.

4.2.5 Recap

To summarize the high points of our design, our compiler will

- Provide a set of annotations with which the user can specify each sparse matrix's storage format,
- Provide a mechanism, the black-box protocol, with which the user can extend the set of storage formats that the compiler is able to handle,
- Use a data-centric approach in order to generate sparse implementations that access each of the sparse matrices efficiently, and
- Use "joins" as a means of reasoning about the inter-matrix constraints that must be satisfied by the sparse implementation.

4.3 The relational model

In addition to joins, we can make other connections between aspects of sparse compilation and relational databases. We call these connections the *relational model* of the sparse compilation problem, and this model serves as a framework in which we will build our design. Having this model gives us two primary benefits. First, it gives us a well established vocabulary with which to discuss the issues of sparse compilation. Second, by making these connections with relational databases, it is possible for us to leverage off of techniques that are discussed in the database literature. In this section, we will discuss the four primary connections that this model makes between relational databases and sparse compilation.

4.3.1 Sparse Matrices as Relations

There are many different ways in which a relation can be stored on disk. The most straightforward storage way is to simply lay out the relation's tuples on disk, one after another. The amount of storage needed is proportional to the data stored, which is good, but sequential searching is required to find any particular tuple, which might be bad. We say "might be bad", because it actually depends upon how the database is accessed. If queries to the database usually involve visiting every tuple in any arbitrary order and if tuples are never deleted from the database, then this minimal storage format might be perfectly appropriate.

However, it is much more likely that users of such a database will want to perform more complex operations on the database, like searching or deleting individual tuples. In this case, it is probably appropriate to compute and store indices of particular fields. The choice of indices to compute is best determined by examining the tuples stored in the database and the most frequently evaluated queries.

Relational databases serve as a natural model for describing sparse matrices. Consider: the array indices of a sparse matrix, together with the value, correspond to the fields of a relation, and each non-zero entry of the matrix corresponds to a tuple in the relation. This analogy is illustrated in Figure 4.2. Furthermore, the indexing structure of a sparse matrix correspond to the indices available for a relational database.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} a & & b & \\ & c & d & e \\ f & g & & \\ & & & h \end{pmatrix} \end{array} \longrightarrow \begin{array}{c|c|c} i & j & v \\ \hline 1 & 1 & a \\ 1 & 3 & b \\ 2 & 2 & c \\ 2 & 3 & d \\ 2 & 4 & e \\ 3 & 1 & f \\ 3 & 2 & g \\ 4 & 4 & h \end{array}$$

Figure 4.2: A sparse Matrix and its corresponding relation

4.3.2 Computation as queries

If sparse matrices can be viewed as relations, then sparse computations can be viewed as relational queries. That is, instead of expressing a sparse computation as a set of loops and array accesses, we can express it using the relational calculus or algebra.

For instance, the loop nest for MMM, in which A and B are sparse and C is dense,

```
for i := 1 to n do
    for j := 1 to n do
```

```
        for k := 1 to n do
            C[i, j] := C[i, j] + A[i, k] * B[k, j];
        end do
    end do
end do
```

will have the sparsity guard $BVAL(A[i,k]) \wedge BVAL(B[k,j])$ computed for its body. We can express the iterations that must be performed by the sparse implementation using the following relational calculus expression:

$$\{\langle i,j,k\rangle | 1 \leq i,j,k \leq n \wedge BVAL(A[i,k]) \wedge BVAL(B[k,j])\}$$

If relations are used to model sparse matrices as described above, then this can equivalently be expressed in the relational algebra as,

$$\sigma_{1 \leq i,j,k \leq n} \sigma_{A.k=B.k}(A(i,k,v_A) \times B(k,j,v_B))$$

which can be simplified to

$$\sigma_{1 \leq i,j,k \leq n}(A(i,k,v_A) \bowtie_k B(k,j,v_B))$$

We can take this one step further and express the entire sparse computation as a simple loop nest that enumerates the results of the relational query,

for $\langle i,j,k,v_A,v_B\rangle \in \sigma_{1 \leq i,j,k \leq n}(A(i,k,v_A) \bowtie_k B(k,j,v_B))$ do
    $C[i,j] := C[i,j] + v_A * v_B$;
end do

This representation of the sparse computation is convenient for our use because it is intentional. That is, the sequential for loops and other control structures that were present in the original dense specification have been stripped away, and what remains is a concise description of *what* is to be computed, without any indication of *how* it is to be computed.

4.3.3 Joins

We have already alluded to "joins" as a means of reasoning about constraints between sparse matrices that must be satisfies by the sparse implementation. We introduced this idea as one of "matching" between indices, but if the sparse computation is described in the relational algebra, the connection can be seen more formally. In particular, we can use the equivalence between cross products and joins,

$$R_1 \bowtie_{f_1,f_2,\ldots} R_2 = \pi_{f_1,f_2,\ldots}\sigma_{R_1.f_1=R_2.f_1 \wedge R_1.f_2=R_2.f_2 \wedge \ldots}(R_1 \times R_2)$$

to reformulate a query using the equi-join operator. The advantage of doing this is that, while the cross-product operator can only be implemented using nested loops, the join operator can be implemented using nested loops, but has many more possible implementations as well. Several such implementations were presented in Section 3.2.2.

4.3.4 Compilation as scheduling

When a query is presented to a database system, it is often expressed using cross-products and selections. Consider the following SQL query taken from [120],

```
SELECT NAME
FROM ORDERS, INCLUDES, SUPPLIES
WHERE CUST = 'Zack Zebra'
    AND ORDERS.O# = INCLUDES.O#
    AND INCLUDES.ITEM = SUPPLIES.ITEM;
```

The direct translation of this query into the relational algebra would be

$$\pi_{NAME}\sigma_{\substack{INCLUDES.ITEM=SUPPLIES.ITEM \\ \wedge ORDERS.O\#=INCLUDES.O\#}} \begin{pmatrix} ORDERS \times INCLUDES \\ \times SUPPLIES \end{pmatrix}$$

If the database system were to evaluate this query naively, it would take

$$O(\#ORDERS \times \#INCLUDES \times \#SUPPLIES)$$

time, which is very inefficient. However, a database system has the freedom of reexpress the query using joins and then to find join implementations that complement the structure of each relation.

Query optimization, the process of finding efficient, join-based, evaluations of the user's query, was introduced in Section 3.2. During query optimization, the following operations are performed,

- The query is analyzed to determine what joins are available.
- The order in which the joins will be evaluated is determined. We call this *join scheduling.*
- An implementation strategy is chosen for each join. We call this *join implementation.*
- A set of instructions is produced based upon the scheduling decisions. These instructions, when executed, will evaluate the query. We call this *code generation.*

Given that sparse computations can be described using the relational algebra, it would seem likely that query optimization techniques could be used to generate efficient sparse implementations for these queries. Thus, the bulk of the sparse compilation problem is reduced to the query optimization problem! Since a large amount of effort has been devoted to finding good query optimization algorithms, we find that we have a large body of research to draw upon when designing our compiler by using the relational model.

4.4 Further difficulties

The relational model is a promising approach to sparse compilation. However, we cannot simply take the best techniques from the database literature and use them directly to build a sparse compiler. There are several important differences between sparse computations and relational queries that have to be dealt with when applying the relational model.

4.4.1 Affine constraints

One difficulty involves affine constraints appearing in the query. Suppose that we were to represent the following "stencil" code,

for $i := 2$ to $n-1$ do
  $X[i] := X[i] + \alpha * X[i-1] + \alpha * X[i+1];$
end do

as the query,

for $\langle i, x_v, x'_v, x''_v \rangle \in$
  $\sigma_{2 \leq i \leq n-1}$
  $\sigma_{i = X.i \wedge i-1 = X'.i \wedge i+1 = X''.i}$
  $(X(i, v_X) \times X'(i, v_{X'}) \times X''(i, v_{X''}))$ do
  $v_X := v_X + \alpha * v_{X'} + \alpha * v_{X''};$
end do

where X, X', and X'' represent different occurrences of the same original sparse vector X. We call the predicate of the σ, the *affine constraints* of the query, for obvious reasons. Such affine array access functions are not uncommon in real numerical programs, and they introduces several complications when designing a compiler using the relational model.

How to represent the constraints. The definition of equi-join allows cross-products and selections to be replaced with joins, but only when the selection predicate is a simple equality between two sets of fields. However, the selection predicate in this query is an affine equality of fields. In a computation that traverses an array along its diagonals, the selection predicate will include an affine equality of three fields, perhaps $I.i - I.j = A.d$. In general, the selection predicate can be an affine equality in terms of any number of fields.

In order to handle these sorts of codes, we could choose to use an affine version of the equi-join operator that is defined as,

$$\bowtie_{Af=b} (R_1, R_2, \ldots, R_n) = \sigma_{Af=b}(R_1 \times R_2 \times \cdots \times R_n).$$

where $Af = b$ is a linear system describing the affine constraints between the fields named in vector, f, using the coefficient matrix A and the constant vector b. However, in order to represent sparse computations meaningfully as relational algebra queries, we need to use the natural version of this operator. The reason is that, when more general queries are modeled using the relational algebra, only the natural versions of the outer join operators produce sensible results. Unfortunately, a natural and affine version of each of the outer join operators is very cumbersome to define and use systematically.

Instead of using such operators, we will propose a different mechanism for representing the affine constraints in the query in such a way that the simple, natural and equi- version of all of the join operators can be used.

Discovering nested joins. The loop nest,

```
for i := ...
    for j := ...
        for k := ...
            ... A1[i] ... A2[i + j + k] ... A3[j + k] ... ;
        end do
    end do
end do
```

in which A_1, A_2, and A_3, are sparse vectors, has the following affine constraints,

$$
\begin{aligned}
A_1.p &= I.i \\
A_2.p &= I.i + I.j + I.k \\
A_3.p &= I.j + I.k
\end{aligned}
$$

What are the joins that must be performed in this case? It would be possible to perform a single monolithic join to evaluate this query, but such an approach would not yield an efficient sparse implementation. A better approach might be to decompose the query into a sequence of smaller joins that are nested inside of one another.

We will not go into the details here, but there are many possible sequences of nested joins, and each set corresponds to a different loop nest for enumerating the solutions of the original constraints. For instance, the following loop nest enumerates the solutions of the original affine constraints,

```
for t_1 := ... do
    I.i := t_1; A_1.p := t_1;
    for t_2 := ... do
        I.j := t_2;
        for t_3 := ... do
            I.k := t_3; A_2.p := t_1 + t + 2 + t + 3; A_3.p := t_2 + t_3;
            ...
        end do
    end do
end do
```

From these loops, a set of nested joins can be constructed to evaluate the example query. A different loop nest, which enumerates the same solutions is,

```
for u_1 := ... do
    A_3.p := u_1;
    for u_2 := ... do
        I.j := u_2; I.k := u_1 - u_2;
        for u_3 := ... do
            A_1.p := u_3; A_2.p := u_1 + u_3;
            ...
        end do
    end do
end do
```

and might be used to construct another set of nested joins. In general, a set of affine constraints can give rise to many different loop nests and, hence, many different sets of nested joins. This being the case, how do we determine all of the possible sets of nested joins, and how do we choose one set over another?

Scheduling joins with affine constraints. The textbook descriptions of query optimization do not discuss how joins involving affine constraints should be scheduled. This is not surprising, as the standard formulation of the relational algebra ([36], [37]) does not allow these sorts of constraints to appear in expressions, except in extremely restricted circumstances (simple equalities of two fields on $\bowtie$'s and general affine constraints on σ's. In order to apply the existing query optimization techniques to the problem of sparse compilation, we must first extend them to handle general affine constraints that might appear in sparse computations.

Implementing joins with affine constraints. Another complication is how to implement joins with affine constraints. We need to extend the equi-join implementations to account for non-unit coefficients and non-zero offsets. Suppose, for instance, that we wished to implement a join on two fields, f_1 and f_2, satisfying the constraint $f_1+1 = -2f_2$. If we decided to do this using the sort-merge join, then we must account for the fact that, if the values of f_1 are visited in increasing order, then the values of f_2 must be visited in decreasing order. Also, we need to account for f_1's offset of 1 and f_2's step size of 2.

4.4.2 Hierarchical storage

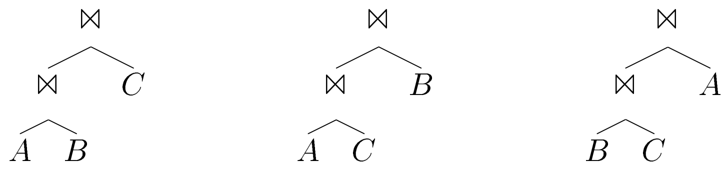
Another difficulty of existing query optimization techniques has to do with the hierarchical nature of some sparse matrix storage formats. Consider the following code for performing MMM:

```
for i := 1 to n do
    for j := 1 to n do
        for k := 1 to n do
            C[i,j] := C[i,j] + A[i,k] * B[k,j];
        end do
    end do
end do
```

Assuming that C has been carefully allocated so that it contains exactly the entries accessed by this computation, then its sparsity structure can be used to narrow the results of the query even further. Thus, the following relational algebra expression can be used to describe the iterations performed by the sparse implementation,

$$A(i,k) \bowtie B(k,j) \bowtie C(i,j)$$

where $\bowtie$ is the natural inner join operator. There are three left deep expression trees that correspond to three different high-level plans for evaluating this query.



(a) (b) (c)

But suppose that A, B, and C are stored in a format that provides efficient access to the non-zeros within each row of a matrix (CRS discussed in Section 7.3.4 is an instance of a storage format that provides such access). In this case, the following strategy can be used to produce an efficient evaluation of the query.

- Enumerate all i's for which there are entries in row i of A and row i of C. For each such i,
- Enumerate all k's for which there is a k entry in row i of A and entries in row k of B. For each such k,
- Enumerate all j's for which there is a j entry in row k of B and a j entry in row i of C.

Each of these steps involves the simultaneous matching of indices and corresponds to a join.

Here is the key observation: while the expression tree form of the high-level plan only made two multiple field joins evident, in this case, there are three single field joins that need to be discovered and scheduled. In order to adapt the existing query optimization methods to our problem, we need to adopt a notation for high-level plans that allows us to break multiple field joins down into single field joins.

4.4.3 Specifying new storage formats

In a conventional database system, a fixed set of database formats and indexing structures are used. As a result, knowledge of all of these storage formats can be hard-coded into the query optimizer. This means that, when deciding in what order and with what strategies to perform joins, the query optimizer has complete knowledge of of the methods in which each storage format can be accessed and the cost of using each method.

Accurate knowledge of the storage formats would appear to be a prerequisite for scheduling a query, even if that query describes a sparse computation. However, we already stated that we want our compiler to be "extensible" and to be able to generate efficient sparse implementations using novel storage formats. This tension, of wanting accurate knowledge of storage formats, and yet being able to add new formats with arbitrary access methods, is not something that we have seen discussed in the database literature.

4.4.4 Fill

Consider the loop nest for copying one sparse vector, V, to another sparse vector, W.

```
for i := 1 to n do
    V[i] := W[i];
end do
```

If, for a particular index i, $V[i]$ is non-zero, but there is no corresponding non-zero entry in W, then one has to be created. This is an instance of fill, which was discussed in Chapter 1. In a database system, a relation would have methods available for creating and deleting tuples. However, many of the most popular sparse matrix formats do not easily allow the insertion or removal of non-zero entries. The only way to insert a non-zero into a sparse vector, for instance, is to allocate completely new storage, and to copy the old values and the new value into the new storage.

Fill can obviously impose an enormous performance penalty if it is not handled carefully. This must be considered when applying the relational model to the sparse compilation problem.

4.4.5 Disjunctive queries

Consider following loop nest, in which V and W are sparse vectors,

```
for i := 1 to n do
    sum := sum + V[i] + W[i];
end do
```

The body of this loop must be executed when either $V[i]$ or $W[i]$ is non-zero. So, the sparsity guard is $BVAL(V[i]) \vee BVAL(W[i])$.

Let us suppose that the relations $V(i, v_V)$ and $W(i, v_W)$ are used to represent the non-zero entries of V and W, respectively. In Section 3.3, we introduced the Outer Join operator, which can be used in this situation to express a relational algebra query that describes this computation,

```
for ⟨i, v_V, v_W⟩ ∈ (V(i, v_V) ↔ W(i, v_W)) do
    sum := sum + v_V + v_W;
end do
```

There are issues that are yet to be resolved. First, under what circumstances should the query that describes a sparse computation use the $\bowtie$ and $\leftrightarrow$, or even $\rightarrow$, operators? Second, how should the null values, ω, that will be bound to v_V and v_W in some case be interpreted?

4.5 Summary

In this chapter, we introduced our approach to sparse compilation. The the major aspects of our design were,

- The user will provide the compiler with the dense specification of the computation together with sparse annotations specifying the format in which each sparse matrix is to be stored.
- The user will be provided with a mechanism for adding new storage formats to the compiler. The mechanism provided by the current compiler is the black-box protocol.
- Our compiler will use a data-centric, and not iteration-centric, approach to sparse compilation.
- Our compiler will be designed around the relational model, which makes the following analogies between sparse compilation and relation databases.
  - Sparse matrices can be modeled with relations.
  - Sparse computation can be expressed using relational queries.

- Inter-sparse matrix constraints can be expressed and implemented using relational joins.
- Sparse compilation can be modeled as query optimization.

In the next chapter, we will discuss how this design will be realized in terms of the operations performed by our sparse compiler.

Acknowledgment. The idea of using viewing sparse computations as relational queries from which joins could be discovered and scheduled using query optimization was first suggested by Kotlyar ([81]).

Chapter 5

Overview of our design

In Chapter 4, we presented the overall goals and the highlights of our approach. In this chapter, we will give an overview of how these are realized in our design. The overview presented here will introduce each of the major components of the Bernoulli sparse compiler, but will not go into their details. The rest of the thesis is devoted to that.

Recall that, the user will provide the compiler with,

- The dense specification of the computation, with annotations on the sparse array declarations indicating how they are to be stored, and
- Black-box modules for each of the sparse matrix storage formats specified in the program.

In this chapter, we will describe the steps that the compiler performs to transform the dense specification into the final sparse implementation. These steps are,

1. Query formulation. The dense specification is analyzed and a query is formed that summarizes the computation.
2. Hierarchy discovery. Each storage format is analyzed and a orderings of its fields, or hierarchies of indices, are constructed.
3. Join discovery. The affine constraints in the query are formed into a linear system, which is then put into a particular echelon form. A sequence of nested joins can then be read from this transformed system.

4. Join Scheduling. The hierarchies of indices and the echelon form of the affine constraints are combined to produce a high-level plan for evaluating the query.

5. Join Implementation. An implementation is selected for each of the joins in the high-level plan to produce a low-level plan.

6. Instantiation. Code for accessing each of the storage formats is obtained from the appropriate black-box and used to construct the final sparse implementation.

We will introduce these step by showing how an example program is transformed as it works its way through the compiler.

5.1 The example

We will take a loop nest for performing the numerical part of MVM as our running example.

```
declare Y : array [1...n] of real;
declare X : array [1...n] of real;
    annotation ( X is stored as a "sparsev" );
declare A : array [1...n, 1...n] of real;
    annotation ( A is stored as a "crs" );

for i := 1 to n do
    for j := 1 to n do
        Y[i] := Y[i] + A[i,j] * X[j];
    end do
end do
```

The annotations tell the compiler that A is stored in crs, the Compressed Row Storage (CRS) format, and that X is stored in sparsev, the sparse vector storage format. Since Y does not have a sparsity annotation, it will be stored in a dense vector.

5.2 The black-boxes

In addition to this dense specification, the user must provide the compiler with the implementation of the crs and sparsev storage formats via the black-box protocol. The protocol seeks to describe a storage format as a database relation, and there are two basic sets of information conveyed by the protocol. The first is a description of the global structure of the relation. This includes such properties as the names of the fields in the relation and how they relate to the indices in a sparse matrix reference. The second set of information is a list of methods that can be used to access the entries of the relation. These are called the *access methods*.

Dense vectors. The compiler provides a black-box for the implementation of dense vectors and arrays, so the user does not have to worry about implementing this. The schema for the dense vector storage format used for Y is $\langle y, v \rangle$, where y and v are the index and value, respectively, of each entry in the vector. The following access methods are available for this format:

- $Y_{\texttt{enum\_y}}[\,] \rightarrow \{y\}$. This notation means, "the access method named `enum_y` takes no arguments and returns of a set of values, y." Each y is an offset at which an entry is stored.

- $Y_{\texttt{lookup\_v}}[y] \rightarrow v$. This notation means, "the access method named `lookup_v` takes one argument, the field y, and returns a single value v. v is the value field at offset y.

Sparse vectors. The sparse vector storage format was introduced in Example 1.1. The schema for the sparse vector storage format used for X is $\langle x, \tilde{x}, v \rangle$. This means that there are three fields, x, the index assigned to an entry, $\tilde{x}$, the offset at which the entry is stored within the storage format, and, v, the value of the entry. We will assume that the following access methods are available for the sparse vector storage format:

- $X_{\texttt{enum\_}\tilde{\texttt{x}}}[\,] \rightarrow \{\tilde{x}\}$. This method will return a set of offsets, $\tilde{x}$, at which entries are stored.

- $X_{\texttt{search\_x}}(x) \rightarrow found?$. This method will search for an entry with a particular index, x. The method returns the boolean $found?$ if an entry for x was found.

- $X_{\texttt{search\_x}}[x] \rightarrow \tilde{x}$. This method returns $\tilde{x}$, the offset at which the result target of $X_{\texttt{search\_x}}(x)$ was found.

- $X_{\texttt{lookup\_x}}[\tilde{x}] \rightarrow x$. This method returns the index, x, of the entry stored at the $\tilde{x}$ offset.

- $X_{\texttt{lookup\_v}}[\tilde{x}] \rightarrow v$. Similarly, for the value, v.

CRS. The CRS format is a sparse matrix storage format, in which each row of the sparse matrix is stored as a sparse vector. The CRS format will be described in more detail in Section 7.3.4. The schema for the CRS format used for A is $\langle a_1, \tilde{a}_2, a_2, v \rangle$, which is the sparse vector schema extended with a row index. The access methods are

- $A_{\texttt{enum\_a}_1}[\,] \rightarrow \{a_1\}$. Return the set of row indices stored in the matrix.

- $A_{\texttt{enum\_}\tilde{\texttt{a}}_2}[a_1] \rightarrow \{\tilde{a}_2\}$. Return the set offsets of all entries within row a_1.

- $A_{\texttt{search\_}\tilde{\texttt{a}}_2}(a_1, a_2) \rightarrow \mathit{found}$? Given an $[a_1, a_2]$ index, search for the $[a_1, a_2]$ entry, and returns true iff it exists.

- $A_{\texttt{search\_}\tilde{\texttt{a}}_2}[a_1, a_2] \rightarrow \tilde{a}_2$. Returns the $\tilde{a}_2$ offset at which $A_{\texttt{search\_}\tilde{\texttt{a}}_2}(a_1, a_2)$ was found.

- $A_{\texttt{lookup\_a}_2}[a_1, \tilde{a}_2] \rightarrow a_2$. Given, $\tilde{a}_2$, the offset of an entry, return the column index of the entry.

- $A_{\texttt{lookup\_v}}[a_1, \tilde{a}_2] \rightarrow v$. Similarly, for the value, v.

5.3 Query formulation

During query formulation a query in the relational algebra is formed from the dense specification that describes the iterations that must be performed and the array elements that will be accessed by the final sparse implementation. Such a query will have several key components,

- a relation for describing the iteration sparse and the affine constraints specified by the array access functions,

- relations for describing the sparse matrix,

- a selection operator whose predicate is the sparsity guard for the loop body,
- the body of the loop.

Query formulation might transform our example loop nest into the following query:

$$
\begin{array}{l}
\textsf{for } \langle i, j, x, y, a_1, a_2, A.v, Y.v, X.v \rangle \in \\
\qquad \sigma_{BVAL(A.v) \wedge BVAL(X.v)} \\
\qquad\quad (I(i, j, x, y, a_1, a_2) \times A(a_1, a_2, v) \times X(x, v) \times Y(y, v)) \textsf{ do} \\
\quad LVAL(Y.v) := RVAL(Y.v) + RVAL(A.v) * RVAL(X.v); \\
\textsf{end do}
\end{array}
$$

In this case, the affine constraints obtained from the array access functions in the dense specification have been folded into the I relation. Also, the expressions of the form $LVAL(R.f)$ and $RVAL(R.f)$ denote the use of the location and value of the $R.f$ field, respectively. Both of these points will be explained in much greater detail in Chapter 8.

5.4 Discovering the hierarchy

The access methods described in Section 5.2 are "flat." That is, they simply tell the compiler what fields can be accessed from what other fields, and they do not directly describe the order in which fields can be efficiently accessed. A *hierarchy of indices* is a total ordering of fields of a storage format that is consistent with at least one order in which the fields can be accessed. When it is clear from context, we will refer to a hierarchy of indices simply as a *hierarchy*. Since this order is not given with the access methods, it must be deduced. Consider the CRS storage format used for A. A quick check of the access methods available for this form reveals that the a_1 field must be accessed before the a_2 field, and that both must be accessed before the v field. This means that its hierarchy is $a_1 \to a_2 \to v$.

At this point, it is not necessary to include the $\tilde{a}_2$ field in the hierarchy because it does corresponds to any of the array indices or its value. Since the hierarchy of indices will be used to resolve constraints with other sparse matrices, only the fields that are "visible" to other sparse matrices need to be ordered. In Chapter 9 we will discuss how these “hidden” fields fit into the hierarchy.

Is there only one hierarchy of indices for a storage format? No. Consider a dense two dimensional array, C, whose access methods might be,

- $C_{\texttt{enum\_r}}[\,] \rightarrow \{r\}$. Return the set of row indices.
- $C_{\texttt{enum\_c}}[\,] \rightarrow \{c\}$. Return the set of column indices.
- $C_{\texttt{lookup\_v}}[r, c] \rightarrow v$. Return the value, v, at index $[r, c]$.

Notice that either the rows can be enumerated with `enum_r` before the columns with `enum_c`, or vice versa. The constraints implied by these access methods only form a partial order on the fields,

$$\begin{tikzcd} r \arrow[dr] & & c \arrow[dl] \\ & v & \end{tikzcd}$$

However, we have said that the hierarchy of indices is a total order, so one of $r \rightarrow c \rightarrow v$ or $c \rightarrow r \rightarrow v$ must be selected. The reader might think that the manner in which the dense array is laid out should determine the total order. That is, if the matrix is stored in a column-major manner, then $c \rightarrow r \rightarrow v$ should be chosen. However, as we will see, the hierarchy of indices is used to guide the join scheduler as it discovers and schedules joins, so the structure of the other sparse matrices in the query must be taken into a account when choosing the hierarchy.

In the running example, the following hierarchies can be deduced for the fields of A, X, and Y,

$$\begin{aligned} A &: a_1 \rightarrow a_2 \rightarrow v \\ X &: x \rightarrow v \\ Y &: y \rightarrow v \end{aligned}$$

5.5 Join Discovery

The next step of the sparse compilation process is to discovery the nested joins that must be performed during the evaluation of the query. This process is complicated by the fact that, unlike relational database systems, which attempt to discover equi-joins from simple equality constraints, we must joins from the set of affine constraints.

Setting up the linear system. In order to discovery the nested joins that must be performed, we start by forming the affine constraints of the query into a single parametric equation,

$$\underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}}_{\mathbf{H}} \underbrace{\begin{pmatrix} t_1 \\ t_2 \end{pmatrix}}_{\mathbf{t}} \underset{=}{=} \underbrace{\begin{pmatrix} I.i \\ I.j \\ A.a_1 \\ A.a_2 \\ Y.y \\ X.x \end{pmatrix}}_{\mathbf{v}},$$

where $(t_1, t_2)^T$ are the parametric variables.

With this formulation, the equi-joins can easily be discovered: a join must be performed between two fields $R_1.f$ and $R_2.f$ if their corresponding rows in $\mathbf{H}$ are equal.

Discovering joins. In the present example, it is evident without forming the linear system that two equi-joins that need to be performed, one between $A.a_1$, $Y.y$ and $I.i$, and the other between $A.a_2$, $X.x$ and $I.j$. However, in general any affine function of the loop indices can be used to access the sparse arrays. Suppose, for instance, that the body of the original dense specification had been,

$$Y[i-10] := Y[i-10] + A[i-j, j] * X[n-j];$$

In this case, the set of joins that must be performed is not so evident. We might form the following parametric equation to describe the solution of the constraints:

$$\underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & -1 \\ 0 & 1 \\ 1 & 0 \\ 0 & -1 \end{pmatrix}}_{\mathbf{H}} \underbrace{\begin{pmatrix} t_1 \\ t_2 \end{pmatrix}}_{\mathbf{t}} + \underbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -10 \\ n \end{pmatrix}}_{\bar{\mathbf{v}}} = \underbrace{\begin{pmatrix} I.i \\ I.j \\ A.a_1 \\ A.a_2 \\ Y.y \\ X.x \end{pmatrix}}_{\mathbf{v}}$$

In this case, what are the joins that need to be performed? As we pointed out in Section 4.4.1, in such a case equi-joins are not sufficient; we need to discover and schedule joins with affine constraints.

If we think of the index fields of the various relations as forming a space, then the affine constraints derived from the array access functions form intersecting hyperplanes within this space. When these intersecting hyperplanes are intersected with the original loop bounds, the resulting region denotes the set of iterations performed and array locations accessed by the original dense computation. The task of join discovery consists of finding a new set of intersecting hyperplanes that denote the same region and that allow efficient joins to be performed. We will call each of the new hyperplanes a *join surface*.

Notation 5.1 (Extracting the row for a single field)

We will use the notation $[\mathbf{M}]_{R.f}$ and $[\mathbf{v}]_{R.f}$ to extract the row of matrix, $\mathbf{M}$, and the element of vector, $\mathbf{v}$, that correspond to the field $R.f$. If the parametric equation, $\mathbf{Ht} + \bar{\mathbf{v}} = \mathbf{v}$, describes the solutions of the affine constraints in the dense specification, then

$$[\mathbf{H}]_{R.f}\mathbf{t} + [\bar{\mathbf{v}}]_{R.f} = [\mathbf{v}]_{R.f} = R.f$$

describes the solution of a field, $R.f$.

Theorem 5.1 *If there exists two integers α and β such that $\alpha[\mathbf{H}]_{R_1.f} + \beta[\mathbf{H}]_{R_2.f} = 0$, then there is an affine constraint between fields $R_1.f$ and $R_2.f$.*

Proof Let $\gamma = \alpha[\bar{\mathbf{v}}]_{R_1.f} + \beta[\bar{\mathbf{v}}]_{R_2.f}$, then,

$$\begin{gathered}
\alpha[\mathbf{H}]_{R_1.f} + \beta[\mathbf{H}]_{R_2.f} = 0 \\
(\alpha[\mathbf{H}]_{R_1.f} + \beta[\mathbf{H}]_{R_2.f})t = 0 \\
\alpha[\mathbf{H}]_{R_1.f}t + \beta[\mathbf{H}]_{R_2.f}t + \alpha[\bar{\mathbf{v}}]_{R_1.f} + \beta[\bar{\mathbf{v}}]_{R_2.f} = \gamma \\
\alpha([\mathbf{H}]_{R_1.f}t + [\bar{\mathbf{v}}]_{R_1.f}) + \beta([\mathbf{H}]_{R_2.f}t + [\bar{\mathbf{v}}]_{R_2.f}) = \gamma \\
\alpha R_1.f + \beta R_2.f = \gamma
\end{gathered}$$

Corollary 5.2 *Any set of rows of* $\mathbf{H}$ *that are multiples of one another are related by a set of such affine constraints.*

Each set of rows of $\mathbf{H}$ that can be related using a set of such affine constraints will form a join surface. Each join surface discovered will become a join in the eventual sparse implementation. In order to perform a join on this surface, these constraints will be used to relate the values of the fields being joined through the parametric variable, t.

Nesting joins. The corollary might lead us to the conclusion that there are three joins surfaces, and hence three joins, in our new example, (1) $I.i$, $Y.y$, (2) $I.j$, $A.a_2$, $X.x$, and (3) $A.a_1$. However, suppose that we schedule the join of $I.i$ and $Y.y$ as the outermost join. Then, in the body of this join, the value of t_1 is fixed. Thus, we can rephrase the parametric system as,

$$\begin{pmatrix} 0 \\ 1 \\ -1 \\ 1 \\ 0 \\ -1 \end{pmatrix} t_2 + \underbrace{\left(\begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} t_1 + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -10 \\ n \end{pmatrix} \right)}_{\bar{\mathbf{v}}'} = \begin{pmatrix} I.i \\ I.j \\ A.a_1 \\ A.a_2 \\ Y.y \\ X.x \end{pmatrix}$$

Since, $\bar{\mathbf{v}}'$ is constant, there is a single join surface involving $I.j$, $A.a_1$, $A.a_2$, and $X.x$.

In the above discussion, we have only shown join surfaces with a single parametric variable. Of course, join surfaces with more than one parametric variable are possible. For instance, the original set of affine constraints can be thought of as a single, huge, join surface whose parametric variables are the original loop indices. While it is certainly possible to use such surfaces when scheduling a query, we will not do so in this thesis. Instead, we will limit ourselves to single variable join surfaces. The reason for doing so is that it is much simpler to discuss the implementations of joins described with a single parametric variable than with multiple parametric variables.

With this restriction, there will be as many surfaces and joins as the column rank of $\mathbf{H}$. However, these joins are not unique. We will later see that joins can be exposed in $\mathbf{H}$ by permuting the rows of $\mathbf{H}$ by $\mathbf{P}$ and then putting $\mathbf{PH}$ into column echelon form. Different permutations, $\mathbf{P}$, will result in different joins being exposed.

5.6 Join Scheduling

But returning to our original example, once the hierarchy of indices of A and X have been discovered, and once the joins have been discovered from the affine constraints, we can produce a high-level plan for the evaluation of the query.

A high-level plan is a sequence of nested joins that

- Is consistent with the order of the discovered joins, and
- Does not violate the order of the fields as given by the hierarchies of indices.

In our running example, such a high-level plan is easy to produce. Recall that the hierarchies of indices are,

$$\begin{aligned} A &: a_1 \to a_2 \to v \\ X &: x \to v \\ Y &: y \to v \end{aligned}$$

and that the order given to the discovered joins is,

$$A.a_1, Y.y, I.i \rightarrow A.a_2, X.x, I.j.$$

A high-level plan that satisfies all of these constraints is,

```
-- Join #1
for Join(A.a_1, Y.y, I.i) do
    -- Join #2
    for Join(A.a_2, X.x, I.j) do
        -- Body
        ...
    end do
end do
```

However, in some cases, the order in which fields are joined and the order in which they can be traversed are different. Consider the loop nest,

```
for i := 1 to n do
    for j := 1 to n do
        sum := sum + A[i,j] * B[j,i];
    end do
end do
```

in which A and B are stored in the CRS format. Suppose that the two arrays are assigned the hierarchy of indices,

$$A : a_1 \rightarrow a_2 \rightarrow v$$
$$B : b_1 \rightarrow b_2 \rightarrow v$$

and that join discovery assigns the following join ordering

$$A.a_1, B.b_2, I.i \rightarrow A.a_2, B.b_1, I.j \rightarrow A.v \rightarrow B.v$$

In this case, there is a conflict between the order in which the fields of B are constrained by the hierarchy of indices and by the join ordering, and the sparse compiler will have to adjust the high-level plan to reconcile these conflicts. One possibility is to break one of the two joins down and to schedule the pieces at different levels of the high-level plan.

```
-- Join #1: first piece.
for Join(A.a_1, I.i) do
    -- Join #2
    for Join(A.a_2, B.b_1, I.j) do
        -- Join #1: second piece.
        for Search(I.i, B.b_2) do
            sum := sum + A.v * B.v
        end do
    end do
end do
```

5.7 Join Implementation

The sparse compiler has to select an efficient implementation for each join in the high-level plan. The result of choosing an implementation for each of the joins and expressing these implementation in terms of the access methods provided by the storage formats is called the *low-level plan.*

In our running example,

- Since Y is dense, Join #1 can trivially be implemented by enumerating the row indices of A with the `enum_i` access method.
- There are several different ways that Join #2 can be implemented, but we will choose to enumerate the non-zero entries in row i of A, using `enum_x̃`, and then to search for an entry with the index i in X using `search_j`.

Here is the resulting low-level plan.

```
-- Join #1
for a_1 ∈ A_enum_a1[ ] do
    y := i := a_1;
    -- Join #2
    for ã_2 ∈ A_enum_x̃[i] do
```

```
        a_2 := A_lookup_a2[ã_2];
        x := j := a_2;
        if X_search_x(x) then
            -- Body
            x̃ := X_search_x[x];
            Y[y] := Y[y] + A_lookup_v[ã_2] * X_lookup_v[x̃];
        end if
    end do
end do
```

5.8 Instantiation

The low-level plan is essentially the final sparse implementation. All that remains is for the occurrences of the access method to be replaced by their actual implementation. Our compiler does this by obtaining these implementations from each of the black-boxes, and then making the appropriate substitutions in the low-level plan. The resulting sparse implementation is,

```
for i := 1 to n do
    for ã_2 := arowptr[i] to arowptr[i + 1] - 1 do
        j := aconind[ã_2];
        (found?, x̃) := BinarySearch(xindx, j);
        if found? then
            Y[i] := Y[i] + avalues[ã_2] * xvalues[x̃];
        end if
    end do
end do
```

5.9 Road map of the rest of the thesis

This concludes the high-level presentation of our approach. This presentation was designed to make the reader familiar with the major components and terminology of our design. The remainder of the thesis is divided into three parts, as follows,

Part II. This portion of the thesis is devoted to flushing out the details of the relational model and the design of the six major components of the compiler.

Part III. The material presented in Part II is sufficient for generating reasonably efficient sparse implementations from many dense specifications. However, it does not produce code that obtain the best performance possible, and it does not handle many of the codes that are to be found in practice. In Part III, we will present methods for obtaining higher performance using conventional dense optimizations and for extending the current design to handle more general codes.

Part IV. In this part, we will give experimental results that demonstrate that our sparse compilation techniques produce code that is competitive with hand-written code and code produced by Bik's sparse compiler. Also, we will discuss some remaining issues and summarize the contributions of this thesis.

Part V. These are the appendices to the thesis. Material that is far too detailed in nature to be placed in the main body of the thesis has been placed here.

Part II

The core techniques

Chapter 6

Preview

In Part I we introduced the problem of sparse compilation and discussed a previous approach. We also introduced the major components of the compiler and briefly described what function they serve and some of the difficulties they must address. Part II of this thesis is devoted to flushing out the details of the basic design and implementation of each of these components. The design described, except as otherwise noted, is implemented in the current Bernoulli sparse compiler.

6.1 Overview of Part II

As we said in Chapter 5, the relational model serves as the basis for the design of our implementation, and there are six basic phases in our sparse compiler design. In Part II, we devote a chapter to elaborating on the relational model and each of these six phases of the compiler.

Chapter 7. We expand upon the idea of viewing sparse matrices as relations. In particular, we describe and discuss the *black-box protocol*, the compiler abstraction of storage formats that allows each to be viewed as relations with access methods.

Chapter 8. We will finish our discussion of how the sparsity guard of a loop nest is computed and then describe how the loop nest can be expressed as a query. We will also discuss certain other preparatory transformations that must be done before query optimization gets underway.

Chapter 9. We will discuss how hierarchies of indices are constructed from the access method provided by each storage format. We will also illustrate some interesting implementation decisions that influence the nature of the hierarchies constructed.

Chapter 10. We will develop a linear algebra framework for discovering and expressing joins. We will show how to form the affine constraints of the query into a single system of linear equations. We will then show how putting this system into a particular echelon form allows us to discover and nest a set of joins.

Chapter 11. We will show how the hierarchies of indices developed in Chapter 9 and the linear algebra framework developed in Chapter 10 can be combined into a single non-deterministic algorithm for producing a high-level plan for evaluating the query. We will also discuss the heuristic that we currently use to make this algorithm deterministic.

Chapter 12. We will show how implementations can be chosen for each of the joins and other operations in the high-level plan. We will show how information obtained from the black-box protocol can be used to direct this process. The result of this process is the low-level plan.

Chapter 13. We will show how the low-level plan is transformed into the final sparse implementation by obtaining the implementation of each access method via the black-box protocol and using these to produce the final sparse implementation.

6.2 Limitations in Part II

In order to make the sparse compiling problem more tractable, we have made several assumptions that restrict the kinds of programs for which our compiler is able to generate efficient schedules. We plan to remove most of these restrictions in the near future, and, where appropriate, we have indicated what work is in progress in these areas.

Perfectly nested, do-any loops. As we said in Chapter 1, a *do-any* loop nest is one whose iterations can safely be executed in any order. Compare this with a *doall* loop nest, whose iterations can safely be executed concurrently. To see the difference, consider the loop for summing the elements of a vector,

```
sum := 0;
for i := 1 to n do
    sum := sum + v[i];
end do
```

The iterations of the i loop may be executed in any order without affecting the final result.[1] However, the iterations of i cannot be executed concurrently, without having to introduce synchronization on the reads and writes to sum.

By limiting our attention to do-any loops, we are limiting our focus to the MVM operation of iterative methods. In our defense,

- This is simplest of the operations, and an obvious place to start.
- The techniques that we will present will be applicable to other do-any loop nests as well. In particular, the techniques discussed in this thesis can be used for the other computations in Class I and II.
- The techniques that we will present here will serve as a foundation upon which techniques for handling loops with dependencies can be built.

Methods for handling imperfectly nested loops with dependencies are discussed in ([83]).

Single format matrices. In Bik's sparse compiler, a set of representatives is found for each sparse matrix, and each representative is assigned its own sparse format. Currently, the sparsity annotations provided by our compiler only allow one storage format to be specified for the entire range of a matrix. Our motivation here is the same as above:

- This is the simplest case,
- It still allows for non-trivial applications, and
- It will serve as a foundation for future work in multiple representative matrices.

[1] This, obviously, assumes that reordering will not introduce roundoff errors that will change the final result. This is a standard assumption in the restructuring compiler literature.

There are some obvious techniques (loop distribution, for example) that can be used to extend the present work to handle multiple format matrices in dependence-free loops. In the case of loops with dependencies, however, it appears that some non-trivial extensions will have to be made to the black-box protocol in order to generate efficient sparse implementations.

Inner join queries. The current implementation will only schedule queries that can be expressed using the inner join operator, $\bowtie$. These include MVM and MMM, but not MMA. However, techniques have been developed for scheduling more general queries and are discussed in Chapter 15.

No fill. The current implementation does not generate code to handle the creation or deletion of non-zero entries that occurs in Class II and IV computations. As we can see in Figure 1.1, there are many non-trivial codes in which fill does not occur, but there are many important computations in which it does. After presenting the core techniques for fill-free codes, we will present extensions to the current design for handling fill in Chapter 16.

6.3 Running Example

In order to illustrate the the ideas throughout Part II, we will show how two example are transformed by each phase of the compiler. Here are the dense specifications of these running examples,

Example 6.1 Dot-product
The following loop nest performs the dot-product of two vectors.

```
sum := 0;
for i := 1 to n do
    sum := sum + X[i] * Y[i];
end do
```

We will assume that X and Y have been declared to be sparse vectors.

Example 6.2 MVM
The following loop nest performs MVM.

```
for i := 1 to n do
    for j := 1 to n do
        Y[i] := Y[i] + A[i,j] * X[j];
    end do
end do
```

We will assume that A has been declared to be a sparse CRS matrix, X has been declared to be a sparse vector, and Y has been declared to be a dense vector.

Chapter 7

A Compiler-Oriented Abstraction of Sparse Matrix Formats

One of the key problems that we had to address while designing our sparse compiler was developing an abstraction for sparse matrices and their many varied storage formats. Such an abstraction must be sufficiently general that the user is able to describe a wide array of novel storage formats and, yet, is not so general that the compiler is unable to generate efficient code for common situations. In Part I, we made two observations that will be the basis for our abstraction. The first is that sparse matrices can be viewed as relational databases. The second is that these relations can be described in terms of the ways in which they can be accessed efficiently or by their access methods. We also said that the interface between the compiler and the modules is called the "black-box protocol." "Black-box" refers to the compiler's view of sparse matrices as abstract objects. "Protocol" refers to the well-defined interface by which information about these is passed from the modules to the compiler and back.

In this chapter, we will discuss the information conveyed by the protocol between the storage format modules and the compiler. We start by discussing how the general aspects of sparse matrices are specified. Then we will discuss how the access methods of each storage format are describes. We will conclude this chapter by describing many commonly used storage formats and discuss how each can be represented within this framework.

In the following discussions, we will use a sparse vector storage format as a running example. In this format, if the vector Y contains nzs_Y non-zeros, then two vectors, ind_Y and val_Y, are used to store the non-zeros of the vector. Each of ind_Y and val_Y are length nzs_Y vectors, with $ind_Y[ii]$ storing the index of the iith non-zero of Y and $val_Y[ii]$ storing its value. An example of the sparse vector storage is shown in Figure 7.1, and below is a loop to compute the dot-product of a dense vector, X, and a sparse vector, Y.

```
sum := 0;
for ii := 1 to nzs_Y do
    i := ind_Y[ii];
    sum := sum + X[i] * val_Y[ii];
```

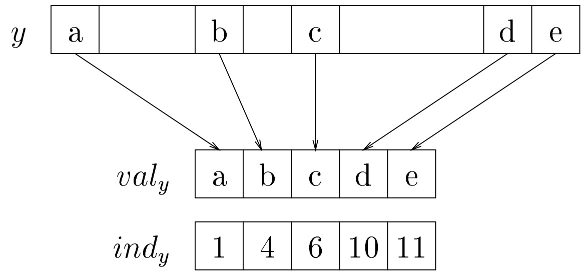


Figure 7.1: Sparse vector storage

7.1 Global properties of the relation

If a sparse matrix is to be modeled as a relation, then there are certain attributes of this relation that be must described to the compiler by the black-box protocol.

7.1.1 The Schema

The first such attribute is the *schema* of the relation, which is the names and types of the fields in the relation. The schema of the sparse vector storage is,

$\langle\ i : \textsf{int},\ ii\text{: } \textsf{int},\ v\text{: } \textsf{real}\ \rangle$

This can be read as follows: a sparse vector is a relation whose tuples have four fields, i, ii, and v, whose values have types, int, int, and real, respectively. If it is clear from context, we will drop the types when presenting schemata.

7.1.2 The Mapping

The second attribute described by the protocol is the *mapping* between an array reference and the fields of the relation. Given a reference to the sparse array, $A[a_1, a_2]$, the mapping specifies which fields of the relation correspond to the row and column indices, a_1 and a_2, and which field contains the value of the entire reference, $A[a_1, a_2]$. We will call the fields corresponding to the array indices the *index fields* and the field corresponding to the array value the *value field*.

Simple Mappings

The mapping for a sparse vector will us that the vector index maps to the i field and that the value of the reference is obtained from the v field.

$$\begin{gathered} y \rightarrow i \\ Y[y] \rightarrow v \end{gathered}$$

Suppose that a storage format whose schema is $\langle i, j, v\rangle$ is used to store a sparse matrix. The mapping for this storage format might tell us that i is the row index, j is the column index, and v is the value field.

$$\begin{gathered} a_1 \rightarrow i \\ a_2 \rightarrow j \\ A[a_1, a_2] \rightarrow v \end{gathered}$$

More Complex Mappings

These two examples illustrate storage formats in which the mapping between the array indices and the index fields is a one-to-one mapping between the components of the sparse vector reference and the field names in the schema. More complicated mappings are possible. Consider the possibilities for a two dimensional array $A[a_1, a_2]$.

Linear mappings. If A's storage format is laid out by diagonal, instead of by row or column, then its schema might be $\langle d, o, v\rangle$, where the d and o fields store the diagonal number and offset of the entries, respectively, and the v field stores their value. In this case, the mapping would be a linear function of the fields,

$$\begin{gathered} a_1 \to o - d \\ a_2 \to o \\ A[a_1, a_2] \to v \end{gathered}$$

Permutation mappings. Another instance of a more complicated mapping occurs in a storage format that uses an arbitrary permutation to map between indices in the array and indices in the storage format. Such a permutation might arise, for instance, as a result of reordering done to reduce the bandwidth of the matrix. Suppose that P is the permutation computed for such a reordering, then the permutation must be applied to both the row and column indices of A in order to obtain the corresponding row and column indices of the storage format:

$$\begin{gathered} P[a_1] \to i \\ P[a_2] \to j \\ A[a_1, a_2] \to v \end{gathered}$$

Why bother?

But why do we bother with these more complicated mappings? Certainly they could be "hidden" within the storage format. Take the diagonal storage example: the linear mapping given above could be pushed into the storage format's access methods, and indices of the storage format could be made to match those of the array.

While this would simply the interface to the storage formats, it is important that the details of these mappings be expose to the compiler, and not hidden within the access methods. Take the linear mapping for the diagonal storage for instance. The linear algebra framework that will be presented in Chapter 10 can use this mapping to ensure that the storage format is accessed in its most efficient manner: by diagonal. If the linear mapping were hidden in the access methods, then there would be no way that this could be done.

Put more concisely, the mappings are exposed to the compiler so that they may be exploited by the compiler.

The current implementation

The current implementation of the sparse compiler only handle simple mappings. While the current implementation of the linear framework discussed in Chapter 10 can handle linear mappings, the current black-box protocol does not provide a mechanism for specifying such mapping. Neither does the black-box protocol provide a mechanism for expressing permutation mappings, nor does the compiler perform the optimizations appropriate for such mappings. Extending the current implementation to correctly handle both of these kinds of mapping is part of the future work.

7.1.3 The Bounds

The third piece of information described by the protocol is the bounds of the indices stored in the relation. That is, if the relation is to be interpreted as an array, what are the bounds of the entries stored in this array? However, if bounds are given as part of each spare array's declaration, then why does the storage format specify bounds as well? The reason is that the bounds of the storage format may not exactly match the bounds of the array that it implements. In other words, the set of array indices specified by the array declaration may be different from the array indices stored.

Consider the following declaration of an array in the dense specification,

declare A : array $[1 \ldots n, 1 \ldots n]$ of real
annotation (A is stored as a "tridiag");

where the tridiag storage format is implements a tridiagonal storage format. In this case, even though the bounds for the entries $A[a_1, a_2]$ as specified by the declaration are,

$$1 \leq a_1, a_2 \leq n$$

the storage format can specify that, in fact, only the entries within the bounds

$$-1 \leq a_1 - a_2 \leq 1, 1 \leq a_1, a_2 \leq n$$

are actually stored. The compiler can use these tighter bounds to prevent useless access of A from occuring.

In addition to expressing bounds tighter than those of the array declaration, the bounds of the array declaration may be tighter than the bounds of the storage format. In this case, the dense specification directs the compiler to produce a sparse implementation in which only a subset of the storage format is to be accessed. In general, the bounds given by the declaration and by the storage format may be equal, contained within one another, intersecting, or even disjoint! It is up to the compiler to ensure that only those indices within the *declared* bounds are ever accesses, and is free to exploit the *stored* bounds when performing optimizations.

7.1.4 The Combining Operator

In each of the sparse matrix storage formats that we consider, there is nothing inherent in their structure to prevent multiple $[i, j]$ entries from being stored. Consider the sparse vector below,

ind_y	1	1	1	2	3	3	4	5	5
val_y	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9

What do the multiple entries for 1, 3, and 5 mean?

Multiple entries often arise in the context of *unassembled* matrices, which a set of multiple $[i, j]$ entries is stored, or more usually generated on the fly. To arrive at the final *assembled* matrix, these entries must be added in order to arrive at the final value for each $[i, j]$ location. In our example this would mean adding each of the multiple 1, 3, and 5 entries to arrive at the final assembled vector,

ind_y	1	2	3	4	5
val_y	0.6	0.4	1.1	0.7	1.7

The fourth piece of information about each relation that is described by the protocol is the operator used to reduce multiple entries to a single assembled entry. This is called the the *combining operator*. If an assembled matrix is stored within a particular instance of the storage format, then multiple entries are not supposed to occur, and a special "no duplicates" operators can be specified. In general, any associate and commutative operator can be used as a combining operator, but in practice only the "addition" and "no duplicates" combining operators are used.

In the current implementation of the black-box protocol, only the "no duplicates" combining operator may be use. It is trivial to add other operators to the protocol, but the additional analysis and transformations that would be described in Section 8.3 will have to be implemented as well.

7.2 Specifying Access Methods

An access method of a relation is a routine for accessing the tuples of a relation. In particular, it is a function with the following signature,

$$\texttt{name} : \langle in\_fields \rangle \rightarrow \{ \langle out\_fields \rangle \}$$

This can be read as, "`name` is an access method whose input is values for $\langle in\_fields \rangle$ and that returns a set of tuples over $\langle out\_fields \rangle$."

7.2.1 The general picture

We can express what an access method does more formally by decomposing its function into two steps. First, the access method selects those tuples from the relation whose $\langle in\_fields \rangle$ match the values of the input parameters. Second, it performs a projection of these tuples onto $\langle out\_fields \rangle$ to form the final results. This can be expressed using the relational algebra, and we can think of the access function `name` as being defined as,

procedure `name`($\langle in\_values \rangle$)
  return $\pi_{\langle out\_fields \rangle} \sigma_{\langle in\_fields \rangle = \langle in\_values \rangle} R$;
end procedure

Thus, at a minimum, an access method can be specified by the following attributes,

A name – $M_{\texttt{foo}}[\ldots]$ refers to the access method with name, `foo`, of the sparse matrix, M.

A list of input fields – The fields that must be specified as arguments to the access method.

A list of output fields – The fields of the values produced by the access method.

7.2.2 Restrictions and additions

In order to make this model more amenable to the compiler, we have to make the following restrictions on access methods the general picture of what constitutes an access method:

- Exactly one output field is allowed for each access method.

and the following additions to the attributes specified for each access method,

A singleton/stream flag. This is a flag indicating the cardinality of the output of the access method. The access method will either produce a singleton tuple of the out_fields, or an arbitrary length stream of tuples.

A cost function. This is an estimate of the cost of executing the access method.

Before going into the details of these restrictions and additions, here are the access methods that might be given for the sparse vector storage format. We will use these in our discussion.

Name	In fields	Out field	Cardinality	Cost
`enum_ii`	$\langle\rangle$	ii	stream	$O(nzs_Y)$
`lookup_i`	$\langle ii\rangle$	i	singleton	$O(1)$
`lookup_v`	$\langle ii\rangle$	v	singleton	$O(1)$
`search_i`	$\langle i\rangle$	ii	singleton	$O(\log nzs_Y)$

In these methods, n is the number of elements in the range of i, and nzs_Y is the number of non-zero elements stored in Y. Note that usually $nzs_Y \ll n$.

7.2.3 Single Output Fields

One might suggest allowing multiple output fields on access methods. For instance, instead of having separate `lookup_j` and `lookup_v` methods, the storage format might provide a single `lookup_j_and_v` method. However, allowing multiple output fields would complicate the design of the join scheduler and the join implementer without adding to the expressive power of the protocol.

7.2.4 Specifying Their Results

The distinction between "singleton" and "stream" results deserves clarification. A method for accessing the value field, v, of a sparse matrix associated with a particular $[i, j]$ entry will, assuming there are no multiple entries, return either a single instance of v or no instances of v if there is no $[i, j]$ entry. Such a method is called the *singleton* method. On the other hand, a method that returns the $[i, j]$ indices of the entries of an array would produce a set of results, whose cardinality is the number of entries stored. This sort of method is called the *stream* method.

Singleton access methods. A singleton access method will return either a single result or no results. The expression $M_{\texttt{name}}(\dots)$ denotes boolean test that will be true if the access method $M_{\texttt{name}}[\dots]$ returns a single result and false if it returns no result. It is an error to invoke the access method $M_{\texttt{name}}[\dots]$ when $M_{\texttt{name}}(\dots)$ is not true. Thus, when singleton access methods are used, they will often appear with the following pattern,

```
if M_name(...) then
    ... M_name[...] ...
end if
```

Stream access methods. A stream method returns a set of results. This set is encoded as an "object" that can be used to enumerate the elements of the set. In what may be slightly confusing terminology, we call the object returned by a stream access method a *stream.* The operations that can be performed on the stream object, h, are

Opening the stream. The following will open and initialize the stream of results:

$$h.init();$$

Checking for the end of stream. The following will return true iff there are more results in the stream:

$$flag := h.valid();$$

Getting the current value. The following will return the current result:

$$out\_field := h.deref();$$

Advancing the stream. The following will advance the stream of the next result:

$$h.incr();$$

Closing the stream. The following will close and finalize the stream:

$h.close()$;

Notation 7.1 (Short hands for stream access)
There are several sorts of loops involving stream access methods that occur so often that it worth giving them a more concise representation.

Enumeration. Here is a loop that will enumerate the results in the stream returned by the access method $A_{\mathtt{enum\_f}}[\ldots]$,

declare h : stream of $A_{\mathtt{enum\_f}}[\ldots]$;
$h.init()$;
while $h.valid$ do
  $f := h.deref()$;
  ...
  $h.incr()$;
end do
$h.close()$;

And, here is the shorthand that we will use instead,

for $f \in A_{\mathtt{enum\_f}}[\ldots]$ do
  ...
end do

Inclusion testing. Here is a loop that will determine whether a particular value is found in the stream returned by the access method $A_{\mathtt{enum\_f}}[\ldots]$,

$found := \#f$;
$h.init()$;
while $h.valid \wedge \neg found$ do
  if $f = h.deref()$ then
    $found := \#t$;
  else
    $h.incr()$;
  end if

```
end do
h.close();
if found then
    ...
end if
```

And, here is the shorthand that we will use instead,

```
if f ∈ A_enum_f[...] then
    ...
end if
```

Eliminating the singleton/stream distinction. Since the singleton result case is actually an instance of the more general stream result case, it might seem appropriate to remove the concept of singleton results from the protocol and to only have stream results. This change would result in a loop being generated for every access to a sparse matrix in order to enumerate the results. In those cases when only a singleton result is ever produced, this would result in significant additional overhead.

Examining the list of access methods given for the sparse vector storage format, the reader will notice that three of the five access methods are marked as “singleton”. If an additional loop were generated for every use of these access methods, the resulting code would have significantly more overhead without any benefit. The bottom line is that the singleton vs. stream distinction occurs often enough in practice and exploiting it makes enough of a difference in performance that it merits being differentiated in the protocol.

7.2.5 The costs

We have attached a cost to each access method. In practice, this does not have to be a precise estimate of the cost; a rough estimate will do. In fact, in the current implementation we distinguish only three costs, “constant-time”, “log-time”, and “linear-time”. In a more sophisticated implementation, we might use this attribute to convey profile information that had been collected from previous uses of the storage-format, as is done in many DBMS's.

7.3 Sparse Matrix Formats

In this section we introduce some sparse matrix storage formats that are representative of the formats that are commonly used in practice. Also, they are used throughout this thesis, so familiarity is important in order to understand the later material. In addition to describing these formats, we will discuss how each can be specified using the black-box protocol discussed above.

In order to make the introduction of these formats more concrete, we will show how the following matrix might be stored in each of the storage formats,

$$\begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{pmatrix} a & & b & \\ & c & d & e \\ f & g & & \\ & & & h \end{pmatrix} \end{array}$$

We also show code for computing the MVM, $Y = A * X$, where X and Y are length n dense vectors and A is a sparse $n \times n$ matrix stored in the format under consideration.

Note: this is not the same as the MVM code that we are using as a running example for Part II of this thesis; in that case X was a sparse vector. Having X dense for the examples in this chapter allows us to focus on what is happening with A.

7.3.1 Coordinate

Coordinate storage is, perhaps, the simplest of the general sparse matrix storage formats. In this format, the non-zeros are stored in a flat "table" without any further indexing structure.

Description. In this format, three vectors, *arowind*, *acolind*, and *avalues* are used to store the row index, column index and value, respectively, of each non-zero element of A. The example matrix in the Coordinate format is shown in Figure 7.2 and the code for computing $Y = A * X$ where A is in Coordinate storage is shown below:

for $k := 1$ to $\#nzs$ do
  $Y[arowind[k]] = Y[arowind[k]] + avalues[k] * X[acolind[k]];$
end do

$avalues$	a	b	c	d	e	f	g	h
$arowind$	1	1	2	2	2	3	3	4
$acolind$	1	3	2	3	4	1	2	4

Figure 7.2: Example matrix in Coordinate storage

Representation. The schema for the Coordinate format is $\langle k, i, j, v \rangle$. A sparse array reference $A[a_1, a_2]$ will be mapped to the fields of the storage format as follows,

$$a_1 \to i$$
$$a_2 \to j$$
$$A[a_1, a_2] \to v$$

That is, the i and j fields will represent the row and column indices of the array, and the v field will represent the array value.

The access methods that might be given for the Coordinate storage format are,

Name	In fields	Out field	Cardinality	Cost
`enum_k`	$\langle\rangle$	k	stream	$O(\#nzs)$
`lookup_i`	$\langle k \rangle$	i	singleton	$O(1)$
`lookup_j`	$\langle k \rangle$	j	singleton	$O(1)$
`lookup_v`	$\langle k \rangle$	v	singleton	$O(1)$

In these methods, n is the number of elements in the range of i and j, and $\#nzs$ is the number of expected number of non-zero elements stored within the array.

7.3.2 Banded storage

The *Banded storage* format is appropriate for sparse matrices in which the non-zero entries lie within a narrow band of diagonals around the center diagonal. It represents a "dense" format, in so much as the row and column indices of each non-zero entry do not have to be explicitly stored. Rather, they are computed from index of the entry within the storage format.

Description. Our sample matrix has a band containing five diagonals, two above the main diagonal and two below. The matrix *avalues* shown in Figure 7.3 is one way that A can be stored in this format.

$$
avalues \qquad
\begin{array}{cc|c|c|c|c|}
 & & \multicolumn{4}{c}{o} \\
 & & \multicolumn{1}{c}{1} & \multicolumn{1}{c}{2} & \multicolumn{1}{c}{3} & \multicolumn{1}{c}{4} \\
\cline{3-6}
 & 2 & \perp & \perp & b & e \\
\cline{3-6}
 & 1 & \perp & 0 & d & 0 \\
\cline{3-6}
d & 0 & a & c & 0 & h \\
\cline{3-6}
 & -1 & 0 & g & 0 & \perp \\
\cline{3-6}
 & -2 & f & 0 & \perp & \perp \\
\cline{3-6}
\end{array}
$$

Figure 7.3: Example matrix in Banded storage

The diagonal index, d, and the offset index, o, are related to the original indices, i and j, by the equalities,

$$
\begin{aligned}
i &= o - d \\
j &= o
\end{aligned}
$$

The code for computing $Y = A * X$ where A is in banded storage is as follows,

```
for d := −#diags to #diags do
    for o := max(d + 1, 1) to min(d + n, n) do
        Y[o − d] = Y[o − d] + avalues[d, o] ∗ X[o];
    end do
```

```
end do
```

Representation. The schema for the Banded storage format is $\langle d, o, v\rangle$. A sparse array reference $A[a_1, a_2]$ will be mapped to the fields of the storage format as follows,

$$a_1 \rightarrow o - d$$
$$a_2 \rightarrow o$$
$$A[a_1, a_2] \rightarrow v$$

The access methods that might be given for the Banded storage format are,

Name	In fields	Out field	Cardinality	Cost
`enum_d`	$\langle\rangle$	d	stream	$O(\#diags)$
`enum_o`	$\langle d\rangle$	o	stream	$O(n)$
`lookup_v`	$\langle d, o\rangle$	v	singleton	$O(1)$

7.3.3 Diagonal Skyline Storage

Diagonal Skyline storage is a variant on Banded storage. It is different in the following aspects,

- A set of an arbitrary diagonals is stored, not simply the diagonals within a particular band, and
- Instead of storing an entire diagonal, only the entries between the first and last non-zero are stored.

This format is designed to have the benefits of a dense storage format, while using less storage than the Banded storage format.

Description. In our sample matrix, there five diagonals that contain non-zero entries.

- The indices of these diagonals are stored in the vector, *adiagind*.
- The values of these diagonals are stored in the vector, *avalues*.

- *adiagptr*[*dd*] points to the first element of the *dd*th diagonal stored in *avalues*. *adiagptr*[*dd* + 1] will point past the last element of this diagonal.
- *adiaglb*[*dd*] − 1 is the offset of the first element stored in the *dd*th diagonal.

The example matrix in the Diagonal Skyline format is shown in Figure 7.4 and the code for computing $Y = A * X$ where A is in Diagonal Skyline storage is shown below,

```
for dd := 1 to #diags do
    d := adiagind[dd];
    for oo := adiagptr[dd] to adiagptr[dd + 1] − 1 do
        o := adiaglb[dd] + oo − adiagptr[dd];
        Y[o − d] = Y[o − d] + avalues[oo] ∗ X[o];
    end do
end do
```

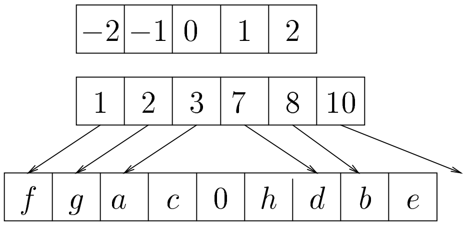


Figure 7.4: Example matrix in Diagonal Skyline storage

Representation. The schema for the Diagonal Skyline format is $\langle d, dd, o, oo, v \rangle$.$\blacksquare$ A sparse array reference $A[a_1, a_2]$ will be mapped to the fields of the storage format as follows,

$$a_1 \rightarrow o - d$$
$$a_2 \rightarrow o$$
$$A[a_1, a_2] \rightarrow v$$

The access methods that might be given for the Diagonal Skyline storage format are,

Name	In fields	Out field	Cardinality	Cost
`enum_dd`	$\langle\rangle$	dd	stream	$O(\#diags)$
`lookup_d`	$\langle dd\rangle$	d	singleton	$O(1)$
`enum_oo`	$\langle dd\rangle$	oo	stream	$O(n)$
`lookup_o`	$\langle dd, oo\rangle$	o	singleton	$O(1)$
`lookup_v`	$\langle dd, oo\rangle$	v	singleton	$O(1)$
`search_o`	$\langle dd, o\rangle$	oo	singleton	$O(1)$

We have started that the cost of `enum_oo` is $O(n)$. This is an instance where the asymptotic cost is overly conservative estimate, and analysis of a set of representative matrices might be able to produce a better quantitative estimate of the cost.

7.3.4 Compressed Row, Column, and Hyperplane Storage

A disadvantage of Coordinate storage is that it does not provide efficient access to the non-zero elements within, say, a particular row of the matrix. The *Compressed Row Storage* (CRS) format was designed to achieve this.

Description. The Compressed Row Storage (CRS),

- Stores entries from the same row contiguously in *avalues* and *acolind*, and
- Replaces *arowind* of the Coordinate format by *arowptr*, a length $n+1$ vector, where $arowptr[i]$ contains the index within *avalues* and *acolind* where the first entry of row i can be found. The $arowptr[n+1]$ entry points one element past the end of *avalues* and *acolind*, and is used simply to record the end of the last row.

The example matrix in the CRS format is shown in Figure 7.5 and the code for computing $Y = A * X$ where A is in CRS storage is shown below,

for $i := 1$ to n do
  for $jj := arowptr[i]$ to $arowptr[i+1] - 1$ do
    $Y[i] = Y[i] + avalues[jj] * X[acolind[jj]];$
  end do
end do

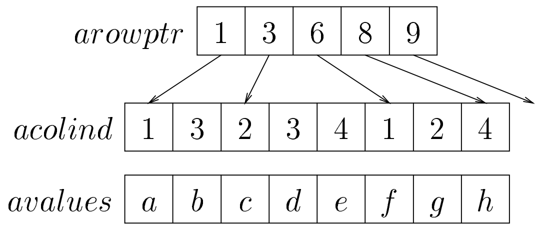


Figure 7.5: Example matrix in CRS storage

Representation. The schema for the CRS format is $\langle i, j, jj, v \rangle$. A sparse array reference $A[a_1, a_2]$ will be mapped to the fields of the storage format as follows,

$$a_1 \to i$$
$$a_2 \to j$$
$$A[a_1, a_2] \to v$$

The access methods that might be given for the CRS storage format are,

Name	In fields	Out field	Cardinality	Cost
`enum_i`	$\langle\rangle$	i	stream	$O(n)$
`enum_jj`	$\langle i \rangle$	jj	stream	$O(N)$
`lookup_j`	$\langle i, jj \rangle$	i	singleton	$O(1)$
`lookup_v`	$\langle i, jj \rangle$	v	singleton	$O(1)$
`search_j`	$\langle i, j \rangle$	jj	singleton	$O(\log \#nzs)$

Variants. Of course, it may be desirable to have efficient access to the nonzeros in a particular column, in which case the Compressed Column Storage (CCS) may be appropriate. CCS,

- Stores entries from the same column contiguously in *avalues* and *arowind*, and

- Replaces *acolind* by *acolptr*, a length $n+1$ vector, in which $acolptr[j]$ contains the index within *avalues* and *arowind* where the first entry of column j can be found.

The example matrix in the CCS format is shown in Figure 7.6 and the code for computing $Y = A * X$ where A is in CCS storage is shown below,

```
for j := 1 to n do
    for ii := acolptr[j] to acolptr[j+1] - 1 do
        Y[arowind[ii]] = Y[arowind[ii]] + avalues[ii] * X[j];
    end do
end do
```

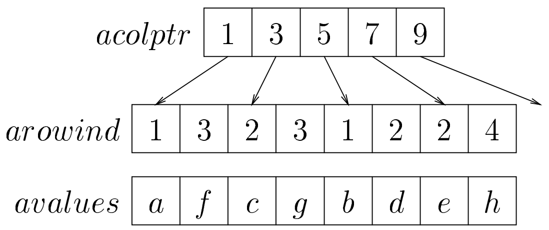


Figure 7.6: Example matrix in CCS storage

The generalization of these two storage formats is the Compressed Hyperplane Storage (CHS) in which an alternative coordinate system (u, v) is used, where u and v are the outer and inner dimensions of the storage, respectively. The original coordinate (i, j) can be obtained by applying some unimodular transformation, T, to (u, v),

$$\begin{pmatrix} i \\ j \end{pmatrix} = T \begin{pmatrix} u \\ v \end{pmatrix}$$

$$T_{\mathrm{CRS}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$T_{\mathrm{CCS}} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

The code for computing $Y = A * X$ where A is in CHS storage is shown below,

```
for u := 1 to n do
    for vv := auptr[u] to auptr[u + 1] − 1 do
        (i, j) := T(u, avind[vv])^T;
        Y[i] = Y[i] + avalues[vv] * X[j];
    end do
end do
```

The representations for the CRS and CCS formats can be easily adapted for the CHS format.

7.3.5 ITPACK

Description. Examining the codes for MVM, it is clear that the number of non-zeros entries within each row of CRS or column of CCS determines the trip count of the computation's innermost loop. For many problems, the number of non-zero entries in either a row or a column is small. For instance, a sparse matrix obtained from a 7 point stencil problem would result in a trip count of 7 for the innermost loop of MVM. This presents a problem for certain high-performance architectures, like vector or superscalar processors that benefit from large inner loop trip counts.

The ITPACK, or ELLPACK, storage format was designed to lengthen the trip count of the innermost loop of MVM. This format is so named because it is the storage format used in those two systems ([77], [107]). In this format, two n by *#diags* matrices, *acolind* and *avalues*, are used to store the column indices and values, respectively, of the non-zero elements. *#diags* is the maximum number of non-zeros within any row of A, and the non-zero entries of $A[i, 1:n]$ are stored in $acolind[i, 1:\#diags]$ and $avalues[i, 1:\#diags]$. Since, not every row of A has $\#diags$ non-zero entries, a special value, say $n+1$, is used to pad rows of $acolind[i, 1:\#diags]$.

avalues

a	b	0
c	d	e
f	g	0
h	0	0

acolind

1	2	5
2	3	4
1	2	5
4	5	5

Figure 7.7: Example matrix in ITPACK storage

The example matrix in the ITPACK format is shown in Figure 7.7 and the code for computing $Y = A * X$ where A is in ITPACK storage is shown below,

```
for jj := 1 to #diags do
    for i := 1 to n do
        if acolind[i, jj] ≠ n + 1 then
            Y[i] = Y[i] + avalues[i, jj] * X[acolind[i, jj]]
        else
            continue jj;
        end if
    end do
end do
```

Notice that, as it has been written, *avalues* and *acolind* would be most efficiently accessed if they were stored in column-major order, as is done in FORTRAN.

Notice, also, the conditional in the inner loop, which checks for the padding value before doing the computation. This conditional will disrupt the control flow of the inner loop and reduce performance on vector and superscalar processors. A work-around is to create a new vector, X', that pads the X vector with an extra entry so that it has length $n+1$, and to initialize this extra element to 0. That way, if we remove the conditional, the padding value $n+1$ will simply access this extra element, and the additional computations that are performed do not alter the final results.

for $jj := 1$ to $\#diags$ do
    for $i := 1$ to n do
        $Y[i] = Y[i] + avalues[i, jj] * X'[acolind[i, jj]]$
    end do
end do

Note that this work-around is not appropriate if there are many padding entries, because the additional flops that will be performed will overshadow anything gained by not performing the conditional. However, if there are many padding entries, the ITPACK storage format will not perform as well as other storage formats anyway. Thus, this work-around is universally used whenever the ITPACK storage format is used.

Representation. The schema for the ITPACK format is $\langle jj, i, j, v\rangle$. A sparse array reference $A[a_1, a_2]$ will be mapped to the fields of the storage format as follows,

$$\begin{aligned} a_1 &\rightarrow i \\ a_2 &\rightarrow j \\ A[a_1, a_2] &\rightarrow v \end{aligned}$$

The access methods that might be given for the ITPACK storage format are,

Name	In fields	Out field	Cardinality	Cost
`enum_jj`	$\langle\rangle$	jj	stream	$O(\#diags)$
`enum_i`	$\langle\rangle$	i	stream	$O(n)$
`lookup_j`	$\langle i, jj\rangle$	j	singleton	$O(1)$
`lookup_v`	$\langle i, jj\rangle$	v	singleton	$O(1)$

In this formulation of the access methods, the conditional that tests for padding entries must be performed within one of the access methods, say, `enum_jj`. The reason is that there is nothing in the protocol for indicating the presence of such entries and to force the compiler to perform the transformation from X to X'. At the present time, we do not have any good ideas for how this might accomplished.

7.3.6 JDiag

Description. An alternative to ITPACK is the Jagged Diagonal (JDiag) storage format. To arrive at this format, we first reorder the rows of A so that the rows with the most non-zero entries appear first, and those with the least appear last,

$$\begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 2 \\ 1 \\ 3 \\ 4 \end{array} & \begin{pmatrix} & c & d & e \\ a & & b & \\ f & g & & \\ & & & h \end{pmatrix} \end{array}$$

Then we store the column indices and values in two matrices, *acolind* and *avalues*, as we did with ITPACK. At this point, instead of actually storing the padding entries, we concatenate the non-zero entries of each column into a vector, and use *adiagptr* to point to the beginning of each column within this linearized storage.

avalues

c	d	e
a	b	0
f	g	0
h	0	0

acolind

2	3	4
1	2	5
1	2	5
4	5	5

Figure 7.8: Example matrix reordered for JDiag

The example matrix in the JDiag format is shown in Figure 7.9 and the code for computing $Y = A * X$ where A is in JDiag storage is shown below,

for $jj := 1$ to $\#diags$ do

  $lb := adiagptr[jj]$
  $ub := adiagptr[jj+1]-1$
  for $i := 1$ to $ub - lb + 1$ do
    $Y[perm[i]] = Y[perm[i]] + avalues[lb+i] * X[acolind[lb+i]]$
  end do
end do

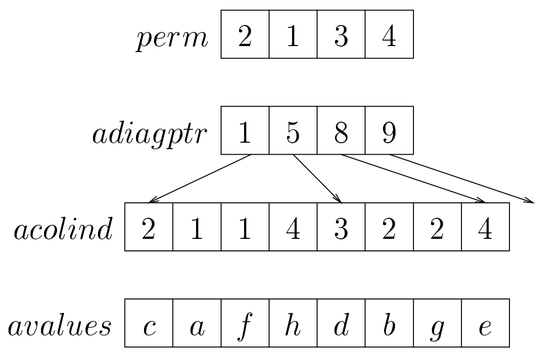


Figure 7.9: Example matrix stored in JDiag

In practice, the permutation that is applied to Y is applied earlier in the program,

for $i := 1$ to n do
  $Y'[i] := Y[perm[i]]$
end do

so that the MVM code can be written as

for $jj := 1$ to $\#diags$ do
  $lb := diagptr[jj]$
  $ub := adiagptr[jj+1]-1$
  for $i := 1$ to $ub - lb + 1$ do
    $Y'[i] = Y'[i] + avalues[lb+i] * X[acolind[lb+i]]$
  end do

```
end do
```

which is easily vectorized.

Representation. The schema for the JDiag format is $\langle jj, i, j, v\rangle$. A sparse array reference $A[a_1, a_2]$ will be mapped to the fields of the storage format as follows,

$$\begin{aligned} a_1 &\rightarrow perm[i] \\ a_2 &\rightarrow j \\ A[a_1, a_2] &\rightarrow v \end{aligned}$$

Note, the storage format cannot assume that the permutation has been applied to Y to obtain Y'. It is the compiler's responsibility to perform this optimization.

The access methods that might be given for the JDiag storage format are,

Name	In fields	Out field	Cardinality	Cost
`enum_jj`	$\langle\rangle$	jj	stream	$O(\#diags)$
`enum_i`	$\langle\rangle$	i	stream	$O(n)$
`lookup_j`	$\langle i, jj\rangle$	j	singleton	$O(1)$
`lookup_v`	$\langle i, jj\rangle$	v	singleton	$O(1)$

7.3.7 BlockSolve

Description. The BlockSolve data structure ([72, 73]) is designed to allow dense regions of a sparse matrix to be stored as dense blocks and to expose parallelism for forward and backward solves. A schematic of a lower triangular matrix stored in BlockSolve format is shown in Figure 7.10.

The outermost level of structure in the BlockSolve format is the *color*. Within each color is a set of *cliques*, which are small dense blocks along the diagonals. Associated with each clique is a set of *inodes*, which are small dense blocks that lie within the same columns as the clique, but are off the diagonal.

The BlockSolve library reorders the matrix to ensure that, when performing a parallel triangular solve, all of the cliques and inodes within a single color can be processed independently, and that synchronization must only

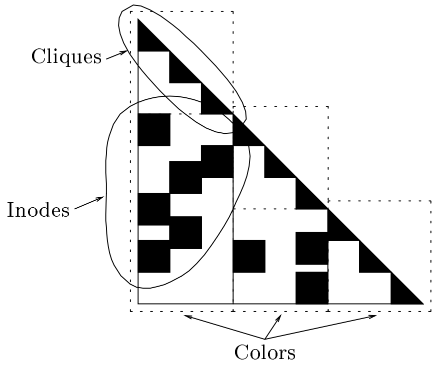


Figure 7.10: BlockSolve storage

occur between one color and the next. The code for computing $Y = A * X$ where A is in BlockSolve storage is shown in Figure 7.11.

Representation. The BlockSolve storage format is problematic for the current compiler, because it violates one of the assumptions stated in Section 4.1. In particular, the BlockSolve storage format is not a single format, but the composition of two formats, the cliques and the inodes. The composition of individual storage formats to construct more complex storage formats is a very interesting and important problem that we do not address in this thesis.

However, we are able to describe the storage format of the cliques and inodes separately. There are several ways to do this; in the experiments that we describe in Section 17.2, we choose to describe the cliques and inodes in the manner in which they are accessed by a single iteration of the *clique* loop. That is, we will describe the storage for (1) a single clique and (2) the list of inodes associated with the clique.

A single clique. A single clique is simply a small dense array along the diagonal of the matrix. The existing dense storage format can be used to describe this.

A list of inodes. The schema for the storage format that describes the list of inodes associate with a single clique will be, $\langle inode, ii, i, jj, j, v \rangle$,

```
for color := 1 to num_colors
    -- Do the color's cliques
    for clique := clique_ptr[color] to clique_ptr[color + 1] − 1 do
        size := clique_sizes[clique];
        index := clique_indices[clique];
        values := clique_values[clique];
        -- MVM for the clique
        for ii := 1 to size do
            for jj := 1 to size do
                Y[index + ii] = Y[index + ii]+
                        values[ii, jj] ∗ X[index + jj]
            end do
        end do
        -- Do the clique's inodes
        for inode := inode_ptr[clique] to inode_ptr[clique + 1] − 1 do
            num_cols := inode_num_cols[inode];
            num_rows := inode_num_rows[inode];
            col_index := inode_col_indices[inode];
            row_indices[1 : num_rows] =
                inode_row_indices[inode, 1 : num_rows]
            values := inode_values[inode]
            -- MVM for the inode
            for ii := 1 to num_rows do
                for jj := 1 to num_cols do
                    i := row_indices[ii];
                    j := col_index + jj;
                    Y[i] := Y[i] + values[ii, jj] ∗ X[j]
                end do
            end do
        end do
    end do
end do
```

Figure 7.11: MVM for the BlockSolve format

where *inode* is the index of each inode in the list, ii and jj are the offset of each row and column, respectively, within an inode, i and j are the index of those rows and columns, respectively, and v is the value of each entry. A sparse array reference $A[a_1, a_2]$ will be mapped to the fields of the storage format as follows,

$$\begin{aligned} a_1 &\to i \\ a_2 &\to j \\ A[a_1, a_2] &\to v \end{aligned}$$

The access methods that might be given for the inode storage format are,

Name	In fields	Out field	Cardinality	Cost
`enum_inode`	$\langle\rangle$	$inode$	stream	$O(\#inodes)$
`enum_ii`	$\langle inode\rangle$	ii	stream	$O(\#rows_{inodes})$
`enum_jj`	$\langle inode\rangle$	jj	stream	$O(\#cols_{inodes})$
`lookup_i`	$\langle inode, ii\rangle$	i	singleton	$O(1)$
`lookup_j`	$\langle inode, jj\rangle$	j	singleton	$O(1)$
`lookup_v`	$\langle inode, ii, jj\rangle$	v	singleton	$O(1)$

7.4 Related work

The first generation of relational data base management systems did not provide an extensible set of storage formats ([62]). However, with the popularization of these system and with their use in an every widening range of applications, the need for allowing user-provided storage formats became clear. Two instances of later systems that provided this sort of extensibility are POSTGRES ([114]) and Starburst ([62]). In both cases, the user can add a new storage format to the DBMS by providing the system with the implementation of a well defined set of access method. Information about the cost of executing these access methods could be provided as well.

In many ways, our approach is similar to these two, but some of the details are different. For instance, the database extension mechanism include methods for inserting and deleting record, which we do not consider except in Chapter 16. Our mechanism, on the other hand, allow simple loops and array references to be distinguished, as will be described in Chapter 14. There is one other difference between the approaches, which will be discussed in Chapter 13.

7.5 Summary

To summarizes, the black-box protocol is used to describe various aspects of the relations used to model the sparse matrix storage formats. These include,

- the schema,
- the mapping between array references and the field of the schema,
- the bounds of the entries stored,
- the combining operator for multiply entries, and
- the access methods provided by the storage format,

and the following for each access method,

- its name,
- the list of input fields required to invoke it,
- the output field produced by invoking it,
- a flag indicating whether a singleton or stream result is produced, and
- an estimate of the cost of invoking it.

Chapter 8

Query Formulation

In this chapter, we will discuss how dense specifications are transformed into relational queries through a process called *query formulation*. We also discuss other transformations that must be made prior to join scheduling. An instance of such transformations are those required when an unassembled matrix is used in a context where its assembled form is required.

8.1 The Query

The primary purpose of query formulation is to extract all of the important information from the original loop nest and to put it in a form that is appropriate for join scheduling. The program representation that is produced by query formulation is called a *query* because of its similarity to a relational database query.

8.1.1 The sample query

We will present a sample query that might be produced by query formulation, and then explain its various components. Consider the following loop nest, which scales a vector, Y and writes the result into X:

for $i := 1$ to n do
    $X[i] := \alpha Y[i]$;
end do

Suppose that X and Y are both sparse. The query produced by query formulation will be,

$$\begin{array}{l} \textsf{for } \langle i, i_X, i_Y, v_x, v_y \rangle \in \\ \qquad \sigma_{BVAL(v_X) \vee BVAL(v_Y)} \\ \qquad\qquad (I(i, i_X, i_Y) \to X(i_X, v_X) \to Y(i_Y, v_Y) \textsf{ do} \\ \quad LVAL(v_X) := \alpha RVAL(v_Y); \\ \textsf{end do} \end{array}$$

We will spend most of the rest of this section discussing the components of this query in detail. At this end of this section, we will precisely specify what constitutes a query.

8.1.2 The array relations

As described in Chapter 7, we will use relations to model sparse matrices. That is, if a two dimensional sparse matrix appears in a loop nest, then a relation with the schema,

$$\langle\ i : \textsf{int},\ j \textsf{: int},\ v \textsf{: real}\ \rangle$$

will be used to model that matrix in the query. There will be a tuple in relation for every non-zero entry that are stored in the corresponding matrix. In other words, there will not be tuples for the entries that are not stored.

Also, if a sparse matrix in the original loop nest is accessed with several different access functions,

$$\begin{array}{l} \textsf{for } i := \ldots \textsf{ do} \\ \qquad \ldots\ A[f(i)]\ \ldots\ A[g(i)]\ \ldots\ A[h(i)]\ \ldots \\ \textsf{end do} \end{array}$$

then, we will use different relations for each distinct access functions. In this example, we will use the relations A_f, A_g, and A_h to represent the different accesses to A. The reason for doing this is that each distinct access function may need to participate in different joins, each of which may need to be implemented differently. This "renaming" does not introduce any problems when there are dependencies, because we have assumed that the original loop nest contained only do-any loops

In the sample query, two array relations are used,

$X(i_X, v_X)$. This relation models the sparse vector, X. This relation has two fields, x, for the index and, v, for the value.

$Y(i_Y, v_Y)$. This relation models the sparse vector, Y.

8.1.3 The iteration space relation

In addition to the relations that describe the sparse arrays, there will be a single relation in the query that describes the iteration space and the array indices that each iteration accesses. In our example query, the relation, I, is defined as,

$$I(i, i_X, i_Y) = \{\langle i, i, i\rangle | 1 \le i \le n\}$$

since the array access functions are $i_X = i$ and $i_Y = i$.

In this example, there is no need to introduce new i_X and i_Y fields; the field i can be used for these purposes. However, in more general computations, non-trivial array access function (e.g., $A[2i + 7j, 3k - 2]$) can be more easily expressed with these additional fields (e.g., $I(i, j, k, a_1, a_2)$ where $a_1 = 2i + 7j$ and $a_2 = 3k - 2$).

8.1.4 The view

We can now define the *view* used by the sparse computation. This view is an aggregate relation in which each tuple contains the loop index values for a single iteration together with the array indices accessed by that iteration and the values stored at these array indices, or ω, if these indices are not stored. To define this view, we use the left outer join operator, introduced in Section 3.3:

$$View(i, i_X, i_Y, v_X, v_Y) = I(i, i_X, i_Y) \rightarrow X(i_X, v_X) \rightarrow Y(i_Y, v_Y)$$

We will assume that the left outer join operator, $\rightarrow$ is left associative. That is,

$$A \rightarrow B \rightarrow C \rightarrow D$$

is grouped as

$$(((A \to B) \to C) \to D)$$

The left outer join operator essentially performs a "search". That is, the result of $R_1 \to R_2$ is all of the tuples of R_1, padded with the corresponding tuples of R_2, if they exists, or ω's, if they do not. So, the composite relation, $View$, can be interpreted as all of the iterations of the original loop nest, together with the array entries that they access, or ω's, for that array values that are accessed but not stored.

It is worth noting at this point that, the reason for extending the iteration space relation, I, with the array indices, i_X and i_Y, is to handle the affine constraints that arise as a result of the array access functions. By folding these constraints into I, we have avoided having to deal with them directly in the algebra. Thus, we can avoid having to define an version of the natural $\to$ operator that satisfies affine, instead of simply equality, constraints. Such an operator would unnecessarily complicates the presentation. The present method allows the affine array functions of the original loop nest to be encoded without defining any additional operators.

Putting it more concisely, by hiding the affine constraints in I we have avoided the need to deal with them explicitly in the relational algebra. However, we have not to eliminated them entirely; these constraints still must be considered when actually scheduling the evaluation of the query.

8.1.5 The sparsity selection

Recall that the sparsity guard is used to prevent useless computation from occuring. In the case of our sample loop nest, the computation is useless only when both $X[i]$ and $Y[i]$ are zero. So, the computation must be performed when either $X[i]$ or $Y[i]$ are non-zero. This sparsity guard can be specified as a predicate of a selection, Hence the term,

$$\sigma_{BVAL(v_X) \vee BVAL(v_Y)} \cdots$$

Instead of actually testing that value fields are 0, we test that they are stored in the sparse matrix. In other words, we test whether or not a value field is ω,

$$BVAL(v) = v \neq \omega$$

The selection predicate is called the *query predicate* and abbreviated as QP. By default, this predicate will be exactly the sparsity guard of the body, $SP(body)$, but additional information about the query may be added as well. This additional information takes the form of logical formulae that are known to be true about the computation but that are not derived by sparsity analysis. Such invariants often arise from program or algorithmic transformations and often cannot be deduced by analysis techniques. Because of this, we will call these invariants *query assertions.* There are many different ways in which query assertions could be specified. In the current compiler, annotations on particular sparse array references are used. The exact details are described in Appendix B.1.

So, the query predicate, QP, will be the conjunction of the sparsity guard, $SP(body)$, and the all of the query assertions.

$$QP = SP(body) \wedge \bigwedge \text{query assertions}$$

This predicate is extremely important during sparse compilation, as any predicate P for which it can be shown that,

$$QP \Rightarrow P$$

is information that the compiler can use to efficiently schedule of the query. This technique is used in a number of places in this thesis.

8.1.6 The looping construct

The relational algebra expressions discussed so far will compute the set of iterations and array entries that will be required in order to perform the computation, but it remains to express that computation. The relational algebra, in its traditional formulation, does not provide procedural constructs for expressing these sorts of computations. Rather, what is is usually done is that a relational algebra notation is "embedded" in an existing conventional language ([2], [34]). We will just use the imperative language that we have been using in the example codes throughout this thesis.

The loop that will enumerate the tuples produced by the relational query will be expressed as,

for $\langle i, i_X, i_Y, v_x, v_y \rangle \in \ldots$ do
    $\ldots$
end do

8.1.7 The body

We have proposed that there be a tuple in the relation, A, for every non-zero entry in the corresponding matrix, A. There must be no null values, ω, stored in an array. However, since we are using the $\rightarrow$ operator to compute the view, ω's can be present in the result of evaluating the query that describes a sparse computation.

Because query evaluation can produce ω's for the value field of a relation, A, we have to worry about what these $\omega's$ mean when the body of the query is evaluated. In order to handle this situation, the following operations are provided for the value field of a relation, A,

$RVAL(v_A)$. Returns an *r-value* of the value field, v_A. If v_A is not ω, then the value is used. If v_A is ω, then the value is not stored in the underlying sparse matrix, so a 0 is used.

$LVAL(v_A)$. Returns an *l-value* reference of the value field, v_A. If v_A is not ω, then the value is stored in the underlying sparse matrix, and its location can be used. If v_A is ω, then the corresponding entry must first be created in the underlying sparse matrix. This new non-zero is referred to as fill, and fill is discussed in Chapter 16.

$BVAL(v_A)$. Returns true iff $v_A \neq \omega$. In other words, returns true if the entry is stored in the underlying sparse matrix, and false otherwise.

In order to obtain the body of the query, the following substitutions are performed on the body of the original loop nest: if A is a sparse matrix, then

- Array references, $A[\ldots]$, are replaced by $LVAL(v_A)$ or $RVAL(v_A)$, depending upon the context in which they appear.

- Boolean tests, $A[\ldots] \neq 0$, are replaced by $BVAL(v_A) \wedge v_A \neq 0$.

Hence, the body of our example query is,

$LVAL(v_X) := \alpha RVAL(v_Y);$

8.1.8 Definition of a query

In general, queries produced by query formulation will have the form,

$\mathsf{for}\ v \in \sigma_{QP}(I \to A_1 \to \cdots \to A_p)\ \mathsf{do}$
  body
$\mathsf{end\ do}$

where,

- A_1 thru A_p will be the relations used to model the sparse matrices that appear in the original loop. Multiple relations will be used to model a single sparse matrix in the event that it is accessed with several distinct access functions.

- I denotes the iteration space relation. More precisely, suppose that $\mathbf{i} \in bounds_{loop}$ is the set of iterations in the original loop nest, and $\mathbf{a}_k = \mathbf{F}_k\mathbf{i} + \mathbf{f}_k$ is the array access function for the kth array access in the original loop nest. Then the iteration space relation is defined as,

$$I(\mathbf{i}, \mathbf{a}_1, \ldots, \mathbf{a}_p) = \{\langle \mathbf{i}, \mathbf{F}_1\mathbf{i} + \mathbf{f}_1, \ldots, \mathbf{F}_p\mathbf{i} + \mathbf{f}_p\rangle | \mathbf{i} \in bounds_{loop}\}$$

- $(I \to A_1 \to \cdots \to A_p)$ is the view on the iteration space and arrays required to perform the computation.

- QP will be the query predicate. This will be the conjunction of the sparsity guard, $SP(body)$, and any query assertions that may be present. $\blacksquare$

- v is a list of all of the fields of the view.

- *body* is the body of the original loop nest, with
  - R-value references, $A_i[\dots]$, replaced by $RVAL(v_{A_i})$.
  - L-value references, $A_i[\dots]$, replaced by $LVAL(v_{A_i})$.
  - Sparsity tests, $A_i[\dots] \neq 0$, replaced by $BVAL(v_{A_i}) \wedge v_{A_i} \neq 0$.

8.2 The Sparsity Guard

The only part of this query that is non-trivial to extract from a loop nest is the sparsity guard, $SP(\mathit{body})$. In this section, we will discuss how the sparsity guard for the body of the query is computed and how it might be simplified.

8.2.1 Sparsity guard for a single assignment

In Section 2.2.1, we showed that the rule for computing $SP(\dots)$, the sparsity guard of a single assignment statement, was

$$SP(var := rhs) = ALC(var) \vee NZ(rhs);$$

where $NZ(rhs)$ is an expression indicating when the rhs is non-zero, and $ALC(var)$ is an expression indicating when storage is allocated for var. There are two optimizations that can be made to this rule.

One case is when the left-hand side is being incremented by the value of the right-hand side. In this case, the statement has the form,

$$var := var + rhs;$$

This statement only needs to be evaluated when the right-hand side is non-zero.

$$SP(var := var + rhs) = NZ(rhs);$$

This is an improvement over the basic algorithm, which would have returned the guard $BVAL(var) \vee NZ(rhs)$.

The other case is when the left-hand side is being scaled by the value of the right-hand side.

$var := var * rhs;$

This statement only needs to be evaluated when the left-hand side is non-zero.

$SP(var := var * rhs) = NZ(var);$

Example 8.1 Assuming that A, B, and C are sparse matrices, here are the sparsity guards for various assignment statements.

$$
\begin{aligned}
&SP(Y[i] := Y[i] + A[i,j] * X[j]) \\
&\quad = BVAL(A[i,j]) \wedge BVAL(X[j]) \\
&SP(C[i,j] := C[i,j] + A[i,k] * B[k,j]) \\
&\quad = BVAL(A[i,k]) \wedge BVAL(B[k,j]) \\
&SP(C[i,j] := A[i,j] + B[i,j]) \\
&\quad = BVAL(C[i,j]) \vee BVAL(A[i,j]) \vee BVAL(B[i,j]) \\
&SP(A[i,j] := \alpha * A[i,j] \\
&\quad = BVAL(A[i,j])
\end{aligned}
$$

8.2.2 Handling more complex query bodies

In general, a query can have more than one simple assignment statement as its body. For instance, suppose that the body of a query contains two assignment statements,

$P1$: $var1 := rhs1;$
$P2$: $var2 := rhs2;$

where P_1 and P_2 are the sparsity guards associated with each statement. Since each assignment statements needs to be evaluated whenever its sparsity guard is true, the entire body of this query needs to be evaluated when either P_1 or P_2 are true.

$$
\begin{aligned}
&P1 \vee P2 : \{ \\
&\quad P1\text{: } var1 := rhs1; \\
&\quad P2\text{: } var2 := rhs2; \\
&\}
\end{aligned}
$$

In [19], Bik provides an attribute grammar for computing the sparsity guards for bodies that include multiple assignment statements, conditionals, and for loops.

8.2.3 Optimizations of complex bodies

Bik also observes that certain loop transformations can be used to reduce the cost of evaluating the sparsity guards of multiple statement bodies. As we will see in Chapter 12, it is generally the case that the more complicated a query's sparsity guard, then the more expensive it is to evaluate the query. Thus, optimizations that can be performed at this point to simplify the guards can have a significant impact on performance.

Loop distribution. Consider the following query in which statements have been annotated with their sparsity guards,

for $v \in \ldots$ do
$\quad BVAL(A[i,j]) \wedge BVAL(B[i,j]) \vee BVAL(P[i,j]) \wedge BVAL(Q[i,j])$: {
$\quad\quad BVAL(A[i,j]) \wedge BVAL(B[i,j])$:
$\quad\quad\quad sum1 := sum1 + RVAL(A[i,j]) + RVAL(B[i,j])$;
$\quad\quad BVAL(P[i,j]) \wedge BVAL(Q[i,j])$:
$\quad\quad\quad sum2 := sum2 + RVAL(P[i,j]) + RVAL(Q[i,j])$;
$\quad$}
end do

If this query is scheduled in its present form, a significant overhead will be paid to implement the body's sparsity guard. If the *loop distribution* transformation ([121]) is applied to the query in order to split it into two queries, then the resulting code is,

for $v \in \ldots$ do
$\quad BVAL(A[i,j]) \wedge BVAL(B[i,j])$:
$\quad\quad sum1 := sum1 + RVAL(A[i,j]) + RVAL(B[i,j])$;
end do
for $v \in \ldots$ do
$\quad BVAL(P[i,j]) \wedge BVAL(Q[i,j])$:
$\quad\quad sum2 := sum2 + RVAL(P[i,j]) + RVAL(Q[i,j])$;
end do

which will be significantly cheaper to evaluate.

Loop fusion. Similarly, if two queries have the same bounds and sparsity guard, then it might make sense to combine them into a single query using *loop fusion* ([121]). Loop fusion applied to,

$$\begin{array}{l}
\textsf{for } v_1 \in \ldots \textsf{ do} \\
\quad BVAL(Z.v) : sum1 := sum1 + RVAL(Z.v) * X[i] \\
\textsf{end do} \\
\textsf{for } v_1 \in \ldots \textsf{ do} \\
\quad BVAL(Z.v) : sum2 := sum1 + RVAL(Z.v) * Y[i] \\
\textsf{end do}
\end{array}$$

where X and Y are dense, will yield,

$$\begin{array}{l}
\textsf{for } v \in \ldots \textsf{ do} \\
\quad BVAL(Z.v) : \{ \\
\quad\quad sum1 := sum1 + RVAL(Z.v) * X[i]; \\
\quad\quad sum2 := sum1 + RVAL(Z.v) * Y[i]; \\
\quad \}
\end{array}$$

which will have half the loop overhead of the original code.

8.2.4 Breaking statements down to simplify guards

When an assignment statement is additive, it can be reduced to a sequence of statements with simpler guards,

$$\begin{array}{l}
A := A1 + A2 + A3 + A4; \\
\quad\downarrow \\
A := 0; \\
A := A + A1; \\
A := A + A2; \\
A := A + A3; \\
A := A + A4;
\end{array}$$

or

$$
\begin{array}{l}
A := A + A1 + A2 + A3 + A4; \\
\quad \downarrow \\
A := A + A1; \\
A := A + A2; \\
A := A + A3; \\
A := A + A4;
\end{array}
$$

This transformation, combined with loop distribution, can transform a single query with a complex sparsity guard into a series of queries, each with a very simple guard. Similar transformations can be performed for other arithmetic operators.

8.2.5 Summary

As a rule of thumb, the more complex the sparsity guard, the more expensive it is to evaluate. This seems to be particularly true with disjunctive sparsity guards. Thus, it make sense to use transformations to reduce a query with a very complex guard into a sequence of queries with simpler guards. But how simple do these new queries have to be? Another way to state this is, when are these optimizations profitable? It seems likely that there is a tradeoff between the complexity of the original query and the overhead introduce by its reduction to a simpler form. We have not explored this tradeoffs at present, but we plan to do so in the future.

In this thesis, we do not assume that any transformations have been performed to reduce the complexity of the guards. In this way, we are forced to address the general problem of scheduling complex queries.

8.3 Checking for the valid use of combining operators

In Section 7.1, we mentioned that one of the properties of a sparse matrix in our relational abstraction is its combining operator. This is the operator used to reduce multiple entries for a single array index into a single entry. We said that the two most commonly used combining operators are "no duplicates", when a matrix does not have multiple entries with the same index, and "addition", for unassembled finite-element matrices. We also said that, in general, any associate and commutative operator could be used as a combining operator.

But consider the following two loop nests, in which A is a sparse matrix whose combining operator is $+$ and C, for simplicity, is dense,

```
-- Loop nest (a)
for i := 1 to n do
    for j := 1 to n do
        C[i, j] := C[i, j] + A[i, j];
    end do
end do
-- Loop nest (b)
for i := 1 to n do
    for j := 1 to n do
        C[i, j] := A[i, j];
    end do
end do
```

After converting these loop nests to queries, we will have,

```
-- Loop nest (a)
for v ∈ ... do
    C[i, j] := C[i, j] + RVAL(A[i, j]);
end do
-- Loop nest (b)
for v ∈ ... do
    C[i, j] := RVAL(A[i, j]);
end do
```

What does $RVAL(A[i, j])$ mean? Does it mean each of the duplicate entries for $A[i, j]$ separately, or does it refer to a single entry after they have been combined?

Since our techniques for scheduling and implementing these queries are data-centric, we will generate code that will execute the bodies of these queries for every entry of A that satisfies the sparsity guard. In other words, our compiler will generate code that will execute the bodies of these loop for each of the duplicate entries of A separately.

Consider the body of loop (a). If this statement is executed with $RVAL(A[i,j])$ corresponding to every entry stored in A, the duplicate entries will not cause problems. This is because the assignment is incrementing $C[i,j]$ be the $A[i,j]$, and the associative and commutative properties of A's combining operator, $+$, make it safe to do this with each of the duplicate entries separately.

However, this approach is not always safe. Consider the body of loop (b). In this case, we are assigning $C[i,j]$ the value of $A[i,j]$, and the correct result is not obtained by writing each of the duplicate entries of A separately. Furthermore, if $A[i,j]$ had appeared on the left-hand side of an assignment, then none of the entries of A would correspond to $LVAL(A[i,j])$.

In general, we can safely handle a matrix with a non-trivial combining operator, by first using the operator to "assemble" the matrix and then using the assembled result in the actual computation. In the case of loop (b), we can rewrite the original loop nest to assemble A before performing query formulation:

```
for i := 1 to n do
    for j := 1 to n do
        Â[i, j] := 0;
    end do
end do
for i := 1 to n do
    for j := 1 to n do
        Â[i, j] := Â[i, j] + A[i, j];
    end do
end do
for i := 1 to n do
    for j := 1 to n do
        C[i, j] := Â[i, j];
    end do
end do
```

Here are the two criteria that should be used in deciding whether to assemble a matrix or not:

- If the nature of the computation requires a matrix to be assembled, then the compiler must generate code to assemble it.

- If the computation can be performed without assembling a matrix, the compiler should not.

The first criteria is necessary in order for the compiler to produce correct code. However, the reasons for the second are a less obvious, since it is certainly possible to construct situations in which better performance would be obtained for a computation by assembling a matrix.

We choose to avoid assembling matrices wherever possible, because we assume that, since the user has specified that a matrix is to be stored unassembled, they have done so for a good reason. For instance, in many matrix-free methods, which have a form similar to loop (a), assembling A would require a prohibitively large amount of memory. Without having access to the actual values stored in A, there is no way for the compiler to detect such a situation, so the compiler should just leaves such a matrix unassembled. Moreover, if the user want the compiler to assemble a matrix, then the user can insert the following into their code to force it to occur,

```
-- A is unassembled. B is assembled.
for i := 1 to n do
    for j := 1 to n do
        B[i,j] := 0;
    end do
end do
for i := 1 to n do
    for j := 1 to n do
        B[i,j] := B[i,j] + A[i,j];
    end do
end do
-- B is used in the computation instead of A.
```

The transformations done to handle matrix assembly must be performed as part of query formulation, because they need to be performed prior to join scheduling.

8.4 The Complications of Fill

There is another issue that must be addressed during query formulation, and that is, what do we do when there is no storage allocated for a left-hand side reference to a sparse matrix? If it is assumed that var in the simple assignment statement $var := rhs$ is sparse, then the sparsity guard for the statement is $BVAL(var) \vee NZ(rhs)$, and there are three conditions under which the assignment is executed,

- $BVAL(var) \wedge NZ(rhs)$. In this case, rhs generates a non-zero value, and an entry exists to store the value in the var reference. There is no need to change the storage allocated for var in this case.

- $\neg BVAL(var) \wedge NZ(rhs)$. In this case, rhs generates a non-zero value, but there is no entry for the var reference in which to store the value. In this case, we need to generate code to *create* an entry before storing the value. This creation of non-zero values is referred to as *fill* in the sparse matrix literature.

- $BVAL(var) \wedge \neg NZ(rhs)$. In this case, rhs generates a zero value. Instead of storing the zero into the var reference, it is desirable to *remove* this entry from the var reference. This deletion of zero values is referred to as *annihilation* in the sparse matrix literature.

The compiler must determine when fill and annihilation can occur and then generate code to handle it. In this section, we will discuss the tests that the compiler can use for detecting these situations. In Chapter 16, we will discuss various strategies and techniques that can be used to handle with fill and annihilation.

The conditions under which fill or annihilation can occur are as follows: given the assignment $var := rhs$, where var is a sparse array and whose sparsity guard is P,

- Fill will occur only if $P \wedge \neg BVAL(var) \wedge NZ(rhs)$, and

- Annihilation will occur only if $P \wedge BVAL(var) \wedge \neg NZ(rhs)$.

If the compiler cannot prove that either fill or annihilation will *not* occur, then the compiler must generate code to handle these cases, because they *might* occur.

Consider fill and annihilation in the three forms of assignment discussed above:

- Simple assignment, $P = BVAL(var) \vee NZ(rhs)$.

– Fill:

$$\begin{aligned} &P \wedge \neg BVAL(var) \wedge NZ(rhs) \\ &\quad = (BVAL(var) \vee NZ(rhs)) \wedge \neg BVAL(var) \wedge NZ(rhs) \\ &\quad = \neg BVAL(var) \wedge NZ(rhs) \end{aligned}$$

– Annihilation:

$$\begin{aligned} &P \wedge BVAL(var) \wedge \neg NZ(rhs) \\ &\quad = (BVAL(var) \vee NZ(rhs)) \wedge BVAL(var) \wedge \neg NZ(rhs) \\ &\quad = BVAL(var) \vee NZ(rhs) \end{aligned}$$

Therefore, fill and annihilation might both occur.

- Increment, $P = NZ(rhs)$.

  – Fill:

$$\begin{aligned} &P \wedge \neg BVAL(var) \wedge NZ(rhs) \\ &\quad = NZ(rhs) \wedge \neg BVAL(var) \wedge NZ(rhs) \\ &\quad = \neg BVAL(var) \wedge NZ(rhs) \end{aligned}$$

  – Annihilation:

$$\begin{aligned} &P \wedge BVAL(var) \wedge \neg NZ(rhs) \\ &\quad = NZ(rhs) \wedge BVAL(var) \wedge \neg NZ(rhs) \\ &\quad = \#f \end{aligned}$$

  Therefore, fill might occur, but annihilation will not.

- Scaling, $P = BVAL(var)$.

  – Fill:

$$\begin{aligned} &P \wedge \neg BVAL(var) \wedge NZ(rhs) \\ &\quad = BVAL(var) \wedge \neg BVAL(var) \wedge NZ(rhs) \\ &\quad = \#f \end{aligned}$$

– Annihilation:

$$\begin{aligned} P \wedge BVAL(var) \wedge \neg NZ(rhs) \\ &= BVAL(var) \wedge BVAL(var) \wedge \neg NZ(rhs) \\ &= BVAL(var) \wedge \neg NZ(rhs) \end{aligned}$$

Therefore, annihilation might occur, but fill will not.

Except in Chapter 16, we will assume that fill and annihilation do not occur in any query. Using the tests that were presented above, this requirement can be enforced during query formulation.

8.5 Preallocated storage

When we defined the $LVAL$ placeholder, we said that if the value field of a relation is ω, then a new location must be created in the relation. This corresponds to a fill entry being created in the underlying matrix. As we mentioned above, this topic is discussed in Chapter 16. However, there are many important situations when the non-zero structure of a sparse matrix can be precomputed, the matrix preallocated, and no further manipulation of the matrix structure is required. The two most important of these situations are,

Direct solvers. Sparse direct solvers often have three phases,

Symbolic Factorization. The factorization is performed "symbolically". In other words, 0 and 1 bits are used to represent zero and non-zero floating point values, and bit operations are used in place of floating point operations to compute a conservative approximation of the non-zero structure of the final result.

Allocation. The sparsity computed by the symbolic phase is used to preallocate static storage for the final result and to schedule the actual computation.

The Computation. The actual numerical factorization is performed. Since the storage for the result has been precomputed, non-zero entries do not have to be created during the computation.

Communication buffers. When a sparse program is parallelized for a message passing machine, most vectors and arrays, including dense vectors and arrays, are distributed to the local memories of the nodes of the parallel machine. Communication is used to ensure that non-local values are available on each processor, when they are needed. As we will discuss in Section 18.3.3 the communication buffers can be treated as sparse matrices, and a sparse compiler can be used to generate code for the node programs. What is important to recognize at this point is that, even though they appear as sparse matrices, the communication code generated by the parallelizing compiler ensures that all non-local values accessed by a node have storage allocated within the communication buffers.

The property common to both of these cases is that, even though the matrices in question are sparse, they are preallocated, and whenever the final computation accesses the matrix, the elements accessed are guaranteed to be allocated.

If SP is the sparsity guard for a statement, and $A[f(i,j)]$ is a reference to a sparse matrix that has been preallocated, then for each execution of the statement,

- If the implication $SP \Rightarrow BVAL(A[f(i,j)])$ is true, then no fill will occur to the reference of A.
- If the implication $BVAL(A[f(i,j)]) \Rightarrow SP$ is true, then no annihilation will occur to the reference of A.

In general, these implications are not something that can be discovered by program analysis, so they must be provided to the sparse compiler as query assertions.

8.6 Inner join queries

Suppose that a sparse array reference, $v[k]$, appears in a query. If it can be shown that the sparsity of $v[k]$ dominates the query predicate, in other words, that,

$$QP \Rightarrow BVAL(v[k]),$$

then we will call $v[k]$ a *strong reference*. Otherwise, we will call it a *weak reference*.

Let us consider the class of sparse computations whose queries, as formulated using the techniques discussed above, have the property that any sparse array reference is strong. In this case, following theorem proves that the sparsity guard is conjunctive.

Theorem 8.1 *Assume that one or more sparse array references occur within a query, that all of these references are strong, and that the query's sparsity guard is non-trivial (i.e., not identically true or false). Then, the sparsity guard of this query is the conjunction of some subset of the sparse array references.*

Proof Assume that the sparsity guard has been simplified as much as possible. Since the query's sparsity guard is non-trivial, by construction, it consists of some number of terms of the form, $BVAL(v[k])$, where $v[k]$ is a sparse reference. This can be seen by examining the method used for constructing sparsity guards, given in Section 2.2.1.

It remains to be shown that the guard is conjunctive. Suppose that it is not. If the guard is put into Disjunctive Normal Form (DNF), then there will be at least two different conjunctive clauses. Assume, without loss of generality, that a term $BVAL(v'[k'])$ appears in the first clause but not the second. Consider a truth assignment for the guard in which the second clause is made true and $BVAL(v'[k'])$ is made false. In this case, the entire guard will evaluate to true because the second clause is true and the guard is in DNF. However, $BVAL(v'[k'])$ is not true. Therefore, $v'[k']$ cannot be a strong reference, which violates our assumption that all of the sparse references were. $\Box$.

This class of sparse computations corresponds to the set of sparse computations that can be described using the natural inner join operator. This is demonstrated by the following theorem:

Theorem 8.2 *Given the following query for a sparse computation,*

for $v \in \sigma_{SP(body)}(I \rightarrow A_1 \rightarrow \cdots \rightarrow A_p)$ do
    body
end do

if every sparse array reference, A_k, is strong, and $SP(body)$ is non-trivial, the following query is equivalent to the original,

for $v \in (I \bowtie A_1 \bowtie \cdots \bowtie A_p)$ do
    $body$
end do

Proof We can show this in two steps. First, we will show that the $\rightarrow$'s can be changed to $\bowtie$'s. Second, we will show that the $\sigma_{SP(body)}$ then becomes redundant and can be eliminated.

- We show that the $\rightarrow$'s can be changed to $\bowtie$'s by induction on the number of sparse array references.

Base case. If there is exactly one sparse array reference, then the query will have the form,

for $v \in \sigma_{SP(body)}(I \rightarrow A_1)$ do
    $body$
end do

The term $I \rightarrow A_1$ can be written as $(I \bowtie A_1) \cup (I \triangleright A_1)$. Since this union is clearly disjoint, the single query can equivalently be rewritten as a sequence of two queries,

for $v \in \sigma_{SP(body)}(I \bowtie A_1)$ do
    $body$
end do
for $v \in \sigma_{SP(body)}(I \triangleright A_1)$ do
    $body$
end do

The second query will execute the iterations for which $SP(body)$ is true, but $A_1.v$ is ω. However, since A_1 is strong, $SP(body)$ will not be true when $A_1.v$ is ω. Therefore, the second query executes no iterations and can be eliminated.

Inductive case. Assume that a query with k $\rightarrow$'s can be changed to $\bowtie$'s. Then, we can show that a query with $k+1$ $\rightarrow$'s can be changed to a $\bowtie$'s by, first, using an argument very similar to the Base case argument to change the rightmost $\rightarrow$ to a $\bowtie$ and, then, using the inductive hypothesis to change the remaining k $\rightarrow$'s to $\bowtie$'s.

- The test, $\sigma_{SP(body)}$, can safely be removed because it is redundant.

  Since, we assume that ω values are not stored in any of the sparse matrices, the result of evaluating the expression $(I \bowtie A_1 \bowtie \ldots \bowtie A_p)$ will not produce tuples containing null values. Therefore, the guard, $BVAL(A_1) \wedge \ldots BVAL(A_p)$ holds for all tuples produced by this expressions. Since Theorem 8.2 tells us that this expression is a conjunction of all of the sparse references in the query, and since the sparsity guard is a conjunction of a subset of these sparse references, the guard enforced by the joins subsumes the sparsity guard. Therefore, the sparsity guard can be safely eliminated.

So, to summarize: the set of computations in which all of the sparse references are strong, and the sparsity guard is non-trivial is the set of computations that can be expressed in the form,

for $v \in (I \bowtie A_1 \bowtie \cdots \bowtie A_p)$ do
    body
end do

Note that this restriction carefully rules out sparse computations in which fill can occur. In these queries, the sparse guard would not be conjunctive, nor would the sparse array reference giving rise to the fill be strong. Note that this restriction does not preclude annihilation, which must be tested for separately.

In Chapter 6, we stated that the current compiler will only handle queries that are composed from $\bowtie$ operators and does not generate code to handle fill and annihilation. Thus, even though the query formulation method described earlier in this chapter can handle general queries, the subsequent phases of the compiler do not. After performing the general query formulation, it is then necessary to identify the queries that can be successfully processed. This is done simply by

- Testing whether or not all of the sparse array references appearing in the query are strong, and

- Testing each of the $LVAL$'s for fill and annihilation.

This may seem to rule out most useful computations, but the following non-trivial computations still fall within this category:

- Sparse dot-product,
- MVM in which the result vector is a dense vector or a preallocated vector, and
- MMM in which the result vector is a dense matrix or a preallocated matrix.

Some interesting computations that do *not* fall into this category are,

- MMA. Since the sparsity guard for this computation is disjunctive, this involves references that appear on the Right-Hand Side (RHS) of an assignment statement that are not strong. In Chapter 15, we will present a method for scheduling queries that involve disjunctive and trivial predicates.
- MMM in which the sparsity of the result must be determined. Since this involves creating fill, the reference on the Left-Hand Side (LHS) of the assignment statement is not strong. In Chapter 16, we will discuss how fill can be handled.
- Forward and backward triangular solves. While all of the references in these computations are strong, the computations contain loop carried dependences, and cannot be safely expressed using the present query notation. Scheduling queries in the presence of dependencies is discussed in ([83]).

8.7 The current implementation

The current compiler implements only part of the material discussed in this chapter. The basic techniques needed to form a query from a loop nest are implemented in a manner similar to what was presented here, but,

- As we have already mention, the current compiler does not attempt to simplify a query's sparsity guard.

- Unassembled matrices cannot occur in the current implementation, because the current black-box protocol allows only the "no duplicates" combining operator.

We plan to overcome these shortcomings in the near future.

8.8 Related work

The analog of query formulation in the database literature is query parsing ([120], [105]). However, in that case, the complications of handling sparsity and fill obviously do not occur.

The idea of automatically computing sparsity guards was first discussed by Bik in [24]. A more refined and complete presentation of his techniques can be found in his thesis ([19]).

8.9 Running Examples

Example 8.2 Dot-product

Query formulation will produce the following query from the dot-product dense specification,

$$\begin{array}{l}
sum := 0; \\
\textsf{for } \langle i, i_X, v_X, i_Y, v_Y \rangle \in \\
\qquad \sigma_{BVAL(v_X) \wedge BVAL(v_Y)} \\
\qquad\quad (I(i, i_X, i_Y) \to X(i_X, v_X) \to Y(i_Y, v_Y)) \textsf{ do} \\
\quad sum := sum + RVAL(v_X) * RVAL(v_Y) \\
\textsf{end do}
\end{array}$$

Since all of the references in the query are strong, this query can be reexpressed using inner joins,

$$\begin{array}{l}
sum := 0; \\
\textsf{for } \langle i, i_X, v_X, i_Y, v_Y \rangle \in \\
\qquad (I(i, i_X, i_Y) \bowtie X(i_X, v_X) \bowtie Y(i_Y, v_Y)) \textsf{ do} \\
\quad sum := sum + RVAL(v_X) * RVAL(v_Y) \\
\textsf{end do}
\end{array}$$

Example 8.3 MVM

Query formulation will produce the following query from the MVM dense specification,

$$
\begin{array}{l}
\textsf{for } \langle i, j, i_A, j_A, v_A, j_X, v_X, i_Y, v_Y \rangle \in \\
\qquad \sigma_{BVAL(v_A) \wedge BVAL(v_X)} \\
\qquad\qquad (I(i, j, i_A, j_A, i_Y, j_X) \rightarrow A(i_A, j_A, v_A) \rightarrow X(j_X, v_X) \\
\qquad\qquad\quad \rightarrow Y(i_Y, v_Y)) \textsf{ do} \\
\quad LVAL(v_Y) := RVAL(v_Y) + RVAL(v_A) * RVAL(v_X); \\
\textsf{end do}
\end{array}
$$

Since all of the references in the query are strong, this query can be reexpressed using inner joins,

$$
\begin{array}{l}
\textsf{for } \langle i, j, i_A, j_A, v_A, j_X, v_X, i_Y, v_Y \rangle \in \\
\qquad (I(i, j, i_A, j_A, i_Y, j_X) \bowtie A(i_A, j_A, v_A) \bowtie X(j_X, v_X) \bowtie Y(i_Y, v_Y)) \textsf{ do} \\
\quad LVAL(v_Y) := RVAL(v_Y) + RVAL(v_A) * RVAL(v_X); \\
\textsf{end do}
\end{array}
$$

8.10 Summary

To summarize the following steps must be performed prior to join scheduling.

Forming the query.

- Sparsity guards must be computed for the body of each loop nest.
- These guards must be combined with the query assertions to form the query predicate.
- Queries of the form shown in Section 8.1.8 are assembled from the loop nests and query predicates.

Checking for fill and annihilation.

- The conditions for fill and annihilation must be tested and ruled out.

Handling the combining operator.

- Queries that involve unassembled matrices must be check to ensure that these computations can be performed without assembling them.
- If matrices must be assembled, then code for performing the assembly must be inserted.

Simplifying the query predicate.

- If appropriate, loop transformation may be made to simplify the query predicate.

Chapter 9

Discovering Hierarchies of Indices

In this chapter, we introduce the concept of a hierarchy of indices of a sparse matrix. These hierarchies are summaries of the ways in which each sparse matrix can be traversed. We will start by discussing how traversals of a relation can be constructed from its access methods. Then we will build upon this idea in order to motivate and formalize the idea of hierarchy of indices. Finally, we will discuss the choice of terms for a hierarchy and how this relates to the design of the join implementer.

9.1 Traversing sparse matrices using access methods

We can use the access methods from a black-box to construct code for enumerating the non-zeros of a sparse matrix, but we cannot actual prove that the code actually performs this task. Since the problem of analyzing such code in order to determine whether or not it enumerates all tuples of a sparse matrix is undecidable, we have to use an indirect means of reasoning about the completeness of these access methods. That is, we will reason about the fields of a sparse matrix that are produced by the access methods.

A field of a relation is the *value field* if the storage format's mapping specifies that the field is used to store the value of array references. A field of a relation is an *index field* if it appears at part of the mapping to array reference and it is not the value field. As we will see, a hierarchy of indices is an ordering of the index and value fields of a relation.

Example 9.1 In a sparse vector, the i field is an index field, the v is a value field, and ii is neither.

If A is a sparse matrix, then a *traversal* of A is a sequence of access methods that meet the following criteria,

- Each access method is an access method of A.
- If we say that the output field of an access method in the sequence is *bound* for the access methods that appear subsequently in the sequence, then no access method appears in a position in which one of its input fields is unbound.
- All of the index fields and the value field of A are bound by access methods in the sequence.

The problem of finding traversals for sparse matrices is called *traversal discovery*.

Example 9.2 Here are but a few of the traversals that can be assigned to the matrices in the running example:

$$\begin{aligned} Y &: \texttt{enum\_y} \rightarrow \texttt{lookup\_v} \\ X &: \texttt{enum\_}\tilde{\texttt{x}} \rightarrow \texttt{lookup\_x} \rightarrow \texttt{lookup\_v} \\ A &: \texttt{enum\_a}_1 \rightarrow \texttt{enum\_}\tilde{\texttt{a}}_2 \rightarrow \texttt{lookup\_a}_2 \rightarrow \texttt{lookup\_v} \end{aligned}$$

The notion of a traversal is an indirect means of ensuring that all of the tuples of A are enumerated. Although the traversal can be used to construct code that will enumerate tuples of A, and it can be shown that these tuples will be well-formed, there is no guarantee that it will produce all (or even any!) tuples of A.

9.1.1 An algorithm for discovering traversals

Traversals are relatively easy to construct from a set of access methods. The key is to recognize when the input fields of an access method become bound by the use of an enclosing access method and that method becomes a candidate for use as the next access method in the sequence.

A non-deterministic algorithm for discovering traversals is illustrated in Figure 9.1 and is defined as a recursive function $FindTraversal$ whose parameters are,

- A is the black-box object for the sparse matrix under consideration,
- $bound$ is a list of the fields of A that have been bound by enclosing calls of $FindTraversal$, and

```
traversal := function FindTraversal(A, bound)
    if index_fields ⊆ bound ∧ value_field ∈ bound then
        -- All the necessary fields are bound.
        traversal := ⟨⟩;
    else
        -- Step 1. Find all ready access methods
        ready_methods := ComputeReadyMethods(A, bound);
        -- Step 2: Choose a ready access method
        am := choose from ready_methods;
        -- Step 3: Recurs
        traversal' := FindTraversal(A, bound ∪ {am.out_field});
        -- Step 4: Add am to the generated sequence
        traversal := am :: traversal'; end function

ready := function ComputeReadyMethods(A, bound)
    ready := φ;
    for am ∈ A.access_methods do
        if am.in_fields ⊆ bound ∧ am.out_field ∉ bound then
            ready := ready ∪ {am};
        end if
    end for
end function
```

Figure 9.1: Non-deterministic algorithm for constructing traversals

Notes:

1. Step 1. An access method is ready when all of its input fields have been bounded by enclosing loops, and its output field has not already been bound by an enclosing loop.

2. Step 2. This is where the non-determinism arises. Any heuristics that might be applied are applied here.

3. Step 3. For the recursion, we add the output field of the selected access method to the list of bound fields and call $FindTraversal$.

4. Step 4. We generate the traversal by prepending the selected access method to the sequence of access methods returned by the recursive call.

9.1.2 Sanity

In Chapter 7, we did not address the question of what constitutes a valid collection of access methods. Consider the following set of access methods for a sparse matrix with the schema, $\langle x, y \rangle$.

Name	In fields	Out field	Cardinality	Cost
$\mathtt{am}_1$	$\langle x \rangle$	y	stream	$O(N)$
$\mathtt{am}_2$	$\langle y \rangle$	x	stream	$O(N)$

There is no way to construct a traversal from these methods. Since finding traversals is at the heart of our scheduling techniques, these access methods, though completely legal, are totally useless for our purposes.

Having presented the algorithm $FindTraversal$, we can give an operational definition of what constitutes a useful set of access methods: the access methods for a sparse matrix, A, are *sane*, if at least one traversal can be discovered using algorithm $FindTraversal$. During compilation, we require that the access methods for every sparse matrix be sane. If they are not, we abort compilation.

9.1.3 Deterministic algorithms

There are many different ways in which we could turn $FindTraversal$ into a deterministic algorithm. For instance, we could develop model for estimating the cost of each traversal and use search techniques to find the minimum cost traversal for each storage format. However, by itself, a deterministic algorithm for finding minimum cost traversals is of limited usefulness. The real problem that we wish to solve is to find a set of traversals for efficiently and simultaneously enumerating the non-zeros of a number of several sparse matrices.

9.2 Hierarchies of indices

As part of constructing the plan for evaluating a query, the join scheduler has to construct nested sequences of fields. Traversals would be an obvious means of represent these sequences; they are, after all, nested sequences of the access methods that can produce these fields. However, each traversal is a complete specification of how a sparse matrix is to enumerated. This is problematic since the join scheduler must somehow leave enough freedom in the plan for the join implementer to be able to select from a wide range of join implementations. If the access methods used to access each sparse matrix are completely specified by the join scheduler, then there is no such freedom.

Hierarchies of indices are summaries that are computed to represent a subsets of the feasible traversals of each sparse matrix. As their name implies, hierarchies provide a nested ordering of the fields of a sparse matrix, just like traversals do. This makes them an appropriate means for the join scheduler to specify the nesting order of the fields in a plan. At the same time, each hierarchy represents a set of feasible traversals, so the join implementer has the range of different access method orderings to choose from in order to implement each join operator.

9.2.1 Partitioning of terms

We start our development of hierarchy of indices by partitioning the fields of each sparse matrix into terms. A *term* will be a set of sparse matrix fields that satisfies these conditions,

- A term may contains exactly one index fields and any number of non-index and non-value fields. Such a term is referred to as a *join term*, because only the terms that contain index fields can possibly participate in joins.

- Otherwise, a term may contains exactly one non-index field.

A *partitioning of terms* is a partitioning of the fields of a sparse matrix into sets, where each set is a term.

Example 9.3 The fields of a sparse vector with the schema $V(ii, i, v)$ can be partitioned into one of the set of terms,

$$\begin{array}{l} \{\{V.i\}, \{V.ii\}, \{V.v\}\} \\ \{\{V.i, V.ii\}, \{V.v\}\} \end{array}$$

The fields of a CRS sparse matrix with the schema $M(i, jj, j, v)$ can be partitioned into one of the set of terms,

$$\begin{array}{l} \{\{M.i\}, \{M.jj\}, \{M.j\}, \{M.v\}\} \\ \{\{M.i, M.jj\}, \{M.j\}, \{M.v\}\} \\ \{\{M.i\}, \{M.jj, M.j\}, \{M.v\}\} \end{array}$$

9.2.2 Hierarchy of indices defined

A total ordering of each term in a partition is called a *term ordering*. We say that a traversal is *consistent* with a term ordering when the order in which the fields are bound within the traversal is consistent with the ordering of the fields within the term ordering. A term ordering for which there is at least one consistent traversal is called a *hierarchy of indices*.

Example 9.4 The following term ordering of fields of a sparse vector, V,

$$\{V.i, V.ii\} \rightarrow \{V.v\}$$

has the following consistent traversals,

`enum_i`, `search_i`, `lookup_v`
`enum_ii`, `lookup_i`, `lookup_v`

and, therefore, constitutes a hierarchy of indices. Here is a different term ordering of the same fields:

$$\{V.i\} \rightarrow \{V.ii\} \rightarrow \{V.v\}$$

which only has one consistent traversal,

`enum_i`, `search_i`, `lookup_v`

and, therefore, also constitutes a hierarchy of indices. The term ordering

$$V.v \rightarrow \{V.i, V.ii\}$$

does not have any consistent traversals, and is not a hierarchy of indices.

Example 9.5 A matrix, B, stored using the ITPACK format (Section 7.3.5) has the access methods,

Name	In fields	Out field	Cardinality	Cost
`enum_jj`	$\langle\rangle$	jj	stream	$O(\#diags)$
`enum_i`	$\langle\rangle$	i	stream	$O(n)$
`lookup_j`	$\langle i, jj\rangle$	j	singleton	$O(1)$
`lookup_v`	$\langle i, jj\rangle$	v	singleton	$O(1)$

might be given the hierarchy of indices,

$$\{B.i\} \rightarrow \{B.j, B.jj\} \rightarrow \{B.v\}$$

However, the following order of the fields,

$$\{B.j, B.jj\} \rightarrow \{B.i\} \rightarrow \{B.v\}$$

is not a the hierarchy of indices of this format. Because the access methods do not allow the $B.j$ field to be accessed before the $B.i$ field, there is no traversal of this format that is consistent with this ordering of the fields.

As can be seen in the examples above, the definition of terms given above ensures that a hierarchy of indices specified a total ordering on the index and value fields of a relation. As we have already seen, there is neither a unique partitioning of terms nor a unique hierarchy of indices for each sparse matrix that appears in the query. This raises the question of how to discover hierarchies of indices. This decision is also tied to the bigger question of join scheduling.

9.2.3 Constructing hierarchies of indices

In Figure 9.2 we give a non-deterministic algorithm for finding a hierarchy of indices for a sparse matrix, A. Once again, we will not bother presenting a deterministic version of this algorithm at this point. We will do so in Chapter 11, as part of join scheduling.

Notation 9.1 (Sequences)

If s_1 and s_2 are sequences, then $s_1 +\!\!+ s_2$ denotes the concatenation of the two sequences.

If a is an element and s is a sequence, then $a :: s$ denotes the sequence obtained by prepending a to s.

If s is a non-empty sequence, then $car(s)$ will denote the first element of s, and $cdr(s)$ will denote the sequence obtained by removing the first element from s.

```
hierarchy := function FindHierarchy(A)
    hierarchy := ε;
    bound := φ;
    while ¬(index_fields ⊆ bound ∧ value_field ∈ bound) do
        let ready_terms = ConstructReadyTerms(A, bound);
        let term = choose from ready_terms;
        hierarchy := hierarchy ++⟨term⟩;
    end do
end function
```

Figure 9.2: Non-deterministic algorithm for finding a hierarchy of indices

9.3 Computing Ready Terms

Examining the $FindHierarchy$ algorithm, the reader might notice that there is nothing to it. The real complexity is to be found in $ConstructReadyTerms$, which performs the partitioning of the fields into terms, determines the sort of hierarchies of indices that will be produced.

The implementation of this routine is governed by the design of the join implementation phase. That is, only those terms will constructed for which the join implementer is able generate code. This is a very important design decision that needs to be stressed: by narrowing the set of terms based upon what the join implementer can handle, we greatly simply the design of join implementer. If we did not do this, then the join implementer would be forced to handle arbitrarily complicated terms. In this remainder of this thesis we will use a single scheme for computing ready terms in order to avoid confusion about which scheme is currently being used. This scheme will be the one used in the current compiler.

But, before precisely specifying the kinds of terms generated by the compiler, we need to say what it means for an access method to be almost ready. An access method is *almost ready with respect to the field* f if it not ready, but if the access method becomes ready when f is added to the set of bound fields. Put another way, the access methods that are almost ready with respect to f are those that become ready once f is bound.

Example 9.6 If none of the fields of a sparse vector are bound, then the v field is almost bound with respect to the ii field, because, once the ii field is bound, the `lookup_v` access method becomes ready.

The notion of term used by the compiler is a slight modification of what has been presented in Section 11.2. In addition to the names of the fields, the terms constructed by our current implementation include information about the access methods that might be used to access the fields of the term. The reason for doing this is that, since these access methods must be computed when the terms are constructed, and since they are required during join implementation, we might as well compute them once and save them as part of the term.

There will be two sorts of terms generated by the current implementation of $ComputeReadyMethods$. They are as follows:

Enum Terms. An enum term contains a stream method. Thus, an enum term will be used to generate a sequence of values. Enum terms will be represented as,

$$\{A.f_1, A.f_2, \ldots\}^*_{am_1,\, am_2,\, \ldots},$$

where $A.f_1, A.f_2, \ldots$ are the fields of the term, and $am_1, am_2, \ldots$ are the access methods associated with the term. There will be three kinds of enum terms, each with a different set of associated access methods.

A Simple Enum Term. This term has a single access method, which is a stream method, and the output field of this method is the only field of the term. Such a term is represented as,

$$\{A.f\}^*_{\mathtt{am}},$$

where am is the solitary stream method, and $A.f$ is its output field. $A.f$ may or may not be an index field. The values of this term could be produced by generating the following loop.

for $A.f \in A_{am}[\ldots]$ do
$\ldots$
end do

An Indexing Enum Term. This term has two access methods. The primary access method, am_1, is a stream method whose output field, $A.f_1$, is not an index field. The secondary access method, am_2 is a singleton method that is almost ready with respect to $A.f_1$ and whose output field, $A.f_2$, is an index field. Such a term is represented as,

$$\{A.f_1, A.f_2\}^*_{am_1,\, am_2}.$$

The values of this term could be produced by generating the following code,

for $A.f_1 \in A_{am_1}[\ldots]$ do

if $A_{am_2}(\ldots, f_1)$ then
    $A.f_2 := A_{am_2}[\ldots, f_1]$;
    ...
end if
end do

A Searching Enum Term. This term has three access methods. The first two access methods and fields are as with the indexing enum term. The third access method, am_3 is a singleton method that is almost ready with respect to $A.f_2$ and whose output field is $A.f_1$. Such a term is represented as,

$$\{A.f_1, A.f_2\}^*_{am_1,\, am_2,\, am_3}.$$

The values of this term could be produced either by generating an indexing loop nest, like,

for $A.f_1 \in A_{am_1}[\ldots]$ do
    if $A_{am_2}(\ldots, f_1)$ then
        $A.f_2 := A_{am_2}[\ldots, f_1]$;
        ...
    end if
end do

or by searching for a known value of the index field, like

if $A_{am_3}(\ldots, f_2)$ then
    $A.f_1 := A_{am_3}[\ldots, f_2]$;
    ...
end if

Lookup Terms. A lookup term's only access method is a singleton method. That is, a lookup term will be used to generate a single value. Lookup terms will be represented as,

$$\{A.f\}^1_{am},$$

where $A.f$ is the field of the term and the output field of the associated singleton method, am. The values of this term could be produced by generating the following code,

if $A_{am}(\dots)$ then
    $A.f := A_{am}[\dots]$;
    $\dots$
end if

When referring to a term, and it is not determined whether the term is an enum term or a lookup term, the notation, $\{A.f, \dots\}^{?}_{\mathtt{am_1},\dots}$ will be used.

The code for $ConstructReadyTerms$, as shown in Figure 9.3, computes a set of ready terms based upon these rules. It should be noted that a field can appear in several ready terms. This is not a violation of the restriction that a field appear in only one term of a partitioning of terms. The set of ready terms is simply a set of choices at a given point in the construction of a hierarchy of indices.

Example 9.7 If no fields are bound in a sparse vector, then these are the ready terms constructed by $ConstructReadyTerms$,

$$\{V.ii\}^{*}_{\mathtt{enum\_ii}}, \{V.ii, V.i\}^{*}_{\mathtt{enum\_ii},\,\mathtt{lookup\_i},\,\mathtt{search\_i}}.$$

Only the later is a join term.

Example 9.8 If ii is bound in a sparse vector, then these are the ready terms,

$$\{V.i\}^{1}_{\mathtt{lookup\_i}}, \{V.v\}^{1}_{\mathtt{lookup\_v}}$$

Only the first is a join term.

Example 9.9 If no fields are bound in a CRS sparse matrix, then the only ready term is $\{A.i\}^{*}_{\mathtt{enum\_i}}$, and it is a join term.

Example 9.10 If the i field is bound in a CRS sparse matrix, then the only ready terms is $\{A.jj, A.j\}^{*}_{\mathtt{enum\_jj},\,\mathtt{lookup\_j},\,\mathtt{search\_j}}$, and it is a join term.

```
ready_terms := function ConstructReadyTerms(A, bound)
    ready_terms := φ;
    for am ∈ ComputeReadyMethods(A, bound) do
        A.f := am.out_field;
        if am is a stream method then
            bound' := bound ∪ {A.f};
            if ∃am' ∈ ComputeReadyMethods(A, bound')
                    ∧am' is a singleton method
                    ∧A.f' = am'.out_field ∧ A.f' ∈ A.index_fields then
                bound'' := bound ∪ {A.f'};
                if ∃am'' ∈ ComputeReadyMethods(A, bound'')
                        ∧ am'' is a singleton method
                        ∧A.f = am''.out_field then
                    ready_terms := ready_terms ∪ {{A.f, A.f'}*_{am, am', am''}};
                else
                    ready_terms := ready_terms ∪ {{A.f, A.f'}*_{am, am'}};
            else
                ready_terms := ready_terms ∪ {{A.f}*_{am}};
        else
            ready_terms := ready_terms ∪ {{A.f}^1_{am}};
    end do
end function
```

Figure 9.3: Algorithm for finding ready terms

9.4 Correctness

This concludes the presentation of the techniques for constructing hierarchies of indices, but several questions remain:

Safety. Does there exist a traversal that is consistent with every hierarchy of indices constructed?

Yes. This can be seen by examining the method of constructing each hierarchy and the three criteria that determine a traversal. Consider the sequence of access methods obtained by the following rules,

- For each term of a hierarchy,
  - if the term has a single access method, `name`, construct the sequence, $\langle$`name`$\rangle$, and
  - if the term is an term with 2 or 3 access methods with the stream method, `enum` and the lookup method, `lookup`, then construct the sequence, $\langle$`enum`, `lookup`$\rangle$.
- Concatenate the sequences produced for each term in the order in which the terms appear in the hierarchy.

This sequence of access methods forms a traversal, since,

- Each of the methods used in the hierarchy are obtained from the black-box of the sparse matrix under consideration.
- By construction, the access methods are not placed into terms until their input fields have been bound by previous access methods.
- All of the index fields and the value field of the sparse matrix are bound by this sequence, since $FindHierarchy$ does not terminate until they are.

Completeness. For every traversal that exists for a sparse matrix, does there exist a hierarchy of indices with which it is consistent?

It depends. In particular, it depends upon the policy and implementation of $ComputeReadyTerms$. Our implementation is not complete. Suppose that a sparse matrix, A, had three access methods,

Name	In fields	Out field	Cardinality
`enum_ii`	$\langle\rangle$	ii	stream
`lookup_i`	$\langle ii\rangle$	i	singleton
`enum_j`	$\langle\rangle$	j	stream

Our implementation of $ComputeReadyTerms$ will produce exactly two terms from these methods,

$$\{A.ii, A.i\}^*_{\texttt{enum\_ii},\,\texttt{lookup\_i}} \qquad \{A.j\}^*_{\texttt{enum\_j}}$$

from which can be constructed the hierarchies of indices,

$$\{A.ii, A.i\}^*_{\texttt{enum\_ii},\,\texttt{lookup\_i}} \rightarrow \{A.j\}^*_{\texttt{enum\_j}}$$
$$\{A.j\}^*_{\texttt{enum\_j}} \rightarrow \{A.ii, A.i\}^*_{\texttt{enum\_ii},\,\texttt{lookup\_i}}$$

Neither of these hierarchies is consistent with the traversal,

$$\texttt{enum\_ii} \rightarrow \texttt{enum\_j} \rightarrow \texttt{lookup\_i}$$

We could make our implementation of $ComputeReadyTerms$ complete by making it non-deterministically choose between constructing simple enum and indexing enum terms. Then it would produce two more terms,

$$\{ii\}^*_{\texttt{enum\_ii}} \qquad \{i\}^1_{\texttt{lookup\_i}}$$

and three more hierarchies of indices,

$$\{ii\}^*_{\texttt{enum\_ii}} \rightarrow \{i\}^1_{\texttt{lookup\_i}} \rightarrow \{j\}^*_{\texttt{enum\_j}}$$
$$\{j\}^*_{\texttt{enum\_j}} \rightarrow \{ii\}^*_{\texttt{enum\_ii}} \rightarrow \{i\}^1_{\texttt{lookup\_i}}$$
$$\{ii\}^*_{\texttt{enum\_ii}} \rightarrow \{j\}^*_{\texttt{enum\_j}} \rightarrow \{i\}^1_{\texttt{lookup\_i}}$$

which does constitute a complete set of hierarchies.

However, even though our current implementation is incomplete, we do not believe that it “loses” interesting traversals. Consider the three extra hierarchies produced by the complete version. The first two are consistent with exactly the same traversals as the original two hierarchies, so they do not actually “add” to the completeness of $ComputeReadyTerms$. The third hierarchy does make the set of hierarchies complete, but the traversals that are consistent with it are not interesting from a query optimization point of view. This is because the query optimization techniques that will be discussed will consider more scheduling options if the `enum_ii` and `lookup_i` methods are listed in a single indexing enum term, and the query optimizer is free to split this term at some later point if it is shown that it is profitable to do so.

To summarize: lumping many access methods into single terms may yield an incomplete set of hierarchies, but this does not prevent the “lost” traversals from being used by the query scheduler.

9.5 Related Work

The concept of hierarchy of indices was introduced in [85]. In that paper, any number of index fields could appear in a term, but in this thesis, a term can only contain one index field. Consider a sparse matrix, A, stored in Coordinate storage format and which has the following access methods,

Name	In fields	Out field	Cardinality	Cost
`enum_k`	$\langle\rangle$	k	stream	$O(\#nzs)$
`lookup_i`	$\langle k\rangle$	i	singleton	$O(1)$
`lookup_j`	$\langle k\rangle$	j	singleton	$O(1)$
`lookup_v`	$\langle k\rangle$	v	singleton	$O(1)$

In [85], the hierarchy of indices given for this storage was,

$$\{A.i, A.j, A.k\} \to \{A.v\}$$

Notice that the first term contains two index fields, i and j. The join implementation phase of the current compiler does not handle joins on more than one index field from each relation,[1] so using single index field terms, we would give two hierarchies of indices to this format,

[1]This is a limitation of the implementation, not of its design, which can easily be extended to handle multiple field joins.

$$\{A.i, A.k\} \rightarrow \{A.j\} \rightarrow \{A.v\}$$
$$\{A.j, A.k\} \rightarrow \{A.i\} \rightarrow \{A.v\}$$

The join scheduler would pick the hierarchy that was most appropriate for a particular query.

The hierarchy discovery problem arises because of the need to accept user-provided access methods, and because these access methods can describe multiple levels of indexing structure. This problem does not appear to be discussed in the database literature, perhaps because multiple level, user-provided, database storage formats do not frequently occur.

9.6 Running Example

Example 9.11 Dot-product

The following hierarchies of indices might be constructed for each of the vectors, X and Y:

$$X : \{X.ii, X.i\} \rightarrow \{X.v\}$$
$$Y : \{Y.ii, Y.i\} \rightarrow \{Y.v\}$$

The following hierarchies of indices might be constructed for the matrix, A, and each of the vectors, X and Y:

Example 9.12 MVM

$$A : \{A.i\} \rightarrow \{A.jj, A.j\} \rightarrow \{A.v\}$$
$$X : \{X.jj, X.j\} \rightarrow \{X.v\}$$
$$Y : \{Y.i\} \rightarrow \{Y.v\}$$

9.7 Summary

In this chapter, we have introduced hierarchies of indices as a means of processing a set of access methods into a structured summary of traversals, prior to join scheduling. We do this for two reasons. First, these summaries allow the join scheduler to focus on discover joins in the query and not have to worry about processing access methods directly. Second, allowing the join implementer to specify the set of acceptable terms imposes some structure on the high-level plans produced by the join scheduler. This greatly simplifies the design of the join implementer.

Chapter 10

The Linear Algebra Framework

In this chapter, we will discuss a linear algebra framework that can be used for discovering a sequence of nested, single parametric variable, join surfaces from a single loop nest. What is interesting and useful about this framework is that it allows the compiler to discover and nest these join surfaces by performing unimodular transformations to an integer linear system instead of having to perform transformations directly to the code. In this case, and in our case, reasonings about a linear system is much simpler than reasoning directly about the code.

We will start with the affine constraints in the query and show how they can be described using a single parametric equation,

$$\mathbf{Ht} + \bar{\mathbf{v}} = \mathbf{v}. \tag{10.1}$$

Next, we will show how a loop nest containing a single join surface can be trivially discovered from this equation. Then, we will generalize this method and show how multiple, nested join surfaces can be discovered by putting $\mathbf{H}$ into column echelon form. We will discuss techniques for arriving at this echelon form, and we present one such algorithm, which will serves as the foundation upon which the join scheduler is built.

10.1 Summarizing affine constraints

Consider the following loop nest, in which X and Y are sparse vectors.

```
for i := 1 to n do
    sum := sum + X[3i − 1] * Y[10 − i];
end do
```

If x and y refer to the index fields on X and Y respectively, then this loop produces the constraints,

$$X.x = 3I.i - 1$$
$$Y.y = 10 - i$$

Recall that these constraints were used to describe the iteration relation that was used in the query,

$$I(i, x, y) = \{\langle i, x, y\rangle | i \in \mathit{bounds} \wedge x = 3i - 1 \wedge y = 10 - i\}$$

This formulation, however, is not the easiest to work with. With the constraints in the form given above, how can we easily determine when entries from different sparse matrices "match"? Also, if we enumerate the entries from a sparse matrix, we need some means of determining whether each entry "matches" an iteration within the original loop nest.

Instead of working with the relational calculus form of the constraints, we will work with an equivalent linear form. Suppose that m array reference of the form, $A_k[\mathbf{F}_k\mathbf{i} + \mathbf{f}_k]$, appear in an n deep loop nest in the dense specification. Then each of these array references will generate a constraint of the form, $A_k.\mathbf{a} = \mathbf{F}_k I.\mathbf{i} + \mathbf{f}_k$, in iteration relation, I. A parametric equation that describes the solution of all of these constraints is given by the single parametric equation,

$$\begin{pmatrix} \mathbf{F}_1 \\ \vdots \\ \mathbf{F}_m \end{pmatrix} I.\mathbf{i} + \begin{pmatrix} \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_m \end{pmatrix} = \begin{pmatrix} A_1.\mathbf{a} \\ \vdots \\ A_m.\mathbf{a} \end{pmatrix} \tag{10.2}$$

However, we will wish to transform this equation, so we will reexpress it in terms of a new set of parametric variables, $\mathbf{t} = (t_1, \ldots, t_n)^T$,

$$\underbrace{\begin{pmatrix} \mathbf{I}_n \\ \mathbf{F}_1 \\ \vdots \\ \mathbf{F}_m \end{pmatrix}}_{\mathbf{H}} \underbrace{\begin{pmatrix} t_1 \\ \vdots \\ t_n \end{pmatrix}}_{\mathbf{t}} \underset{+}{+} \underbrace{\begin{pmatrix} 0 \\ \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_m \end{pmatrix}}_{\bar{\mathbf{v}}} \underset{=}{=} \underbrace{\begin{pmatrix} I.\mathbf{i} \\ A_1.\mathbf{a} \\ \vdots \\ A_m.\mathbf{a} \end{pmatrix}}_{\mathbf{v}} \tag{10.3}$$

where $\mathbf{I}_n$ is the $n \times n$ identity matrix.
The solutions to the constraints in our example can be expressed as,

$$\underbrace{\begin{pmatrix} 1 \\ 3 \\ 4 \end{pmatrix}}_{\mathbf{H}} (t) + \begin{pmatrix} 0 \\ -1 \\ 10 \end{pmatrix} = \begin{pmatrix} I.i \\ X.x \\ Y.y \end{pmatrix}.$$

10.2 Discovering a single join surface

In Section 5.5 it was stated that a join surface is an affine parametric equation between relation fields that is consistent with the original set of affine constraints. We have stated that we will limit our consideration to single variable join surfaces, so each join surface will be given by the parametric equation $\mathbf{h}t + \bar{\mathbf{v}} = \mathbf{v}$, where t is the parametric variable, $\mathbf{h}$ and $\bar{\mathbf{v}}$ are constant vectors, and the entries of $\mathbf{v}$ correspond to the fields to be joined. In order to ensure that this join surface is not degenerate, we will also assume that no entry of $\mathbf{h}$ is zero.

In a sense, what we have done with this formulation is to normalize all of the fields in $\mathbf{v}$ to t. For instance, the solutions to this system can be enumerated by the loop nest,

for $t := lb_t$ to ub_t do
  $\mathbf{v} := \mathbf{h}t + \bar{\mathbf{v}}$;
  ...
end do

where lb_t and ub_t are the bounds on t that are consistent with the original loop bounds. For now we will not worry about how these bounds can be computed; we will discuss this in Section 11.5. We call the equation $\mathbf{h}t + \bar{\mathbf{v}} = \mathbf{v}$ the *join surface form*.

Recall that the affine constraints of our example were formulated as the parametric equation, $\mathbf{Ht} + \bar{\mathbf{v}} = \mathbf{v}$,

$$\underbrace{\begin{pmatrix} 1 \\ 3 \\ 4 \end{pmatrix}}_{\mathbf{H}} \begin{pmatrix} t \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \\ 10 \end{pmatrix} = \begin{pmatrix} I.i \\ X.x \\ Y.y \end{pmatrix}.$$

Since, $\mathbf{H}$ has only one column and $\mathbf{t}$ has only one entry, this system is trivially reformulated in the form $\mathbf{h}t + \bar{\mathbf{v}} = \mathbf{v}$. Thus, we have been able to discover a single join surface between all of the fields of query.

10.3 Discovering multiple join surfaces

In general, $\mathbf{H}$ will have more than one column. In that case, given a system of constraints, $\mathbf{Ht} + \bar{\mathbf{v}} = \mathbf{v}$, how do we discover join surfaces of form $\mathbf{h}t + \bar{\mathbf{v}} = \mathbf{v}$?

10.3.1 The problem

Consider the following MVM code, in which A is accessed by diagonal,

```
for d := 1 − n to n − 1 do
    for o := max(1, 1 − d) to min(n, n − d) do
        i := d + o; j := o;
        Y[i] := Y[i] + A[i, j] ∗ X[j];
    end do
end do
```

Let the relations $I(d, o)$, $A(a_1, a_2)$, $X(x)$, and $Y(y)$ model the iteration space, A, X, and Y, respectively. If we elide i and j, we can extract the following constraints from this code,

$$\begin{aligned} A.a_1 &= I.d + I.o \\ A.a_2 &= I.o \\ Y.y &= I.d + I.o \\ X.x &= I.o \end{aligned}$$

We can express these constraints as either the parametric system,

$$\underbrace{\begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}}_{\mathbf{H}} \underbrace{\begin{pmatrix} t_1 \\ t_2 \end{pmatrix}}_{\mathbf{t}} + \underbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}}_{\bar{\mathbf{v}}} = \underbrace{\begin{pmatrix} A.a_1 \\ A.a_2 \\ Y.y \\ X.x \\ I.d \\ I.o \end{pmatrix}}_{\mathbf{v}}$$

or

$$\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} t_1 + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} t_2 + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} A.a_1 \\ A.a_2 \\ Y.y \\ X.x \\ I.d \\ I.o \end{pmatrix}$$

This is not in the form required for a single join surface.

10.3.2 The solution

What is occuring here is that the solutions of the original system can only be expressed with multiple join surfaces. But, how do we discover these join surfaces? In order to reduce this single parametric equation to a sequence of parametric equations in the join surface form, we need to make certain transformations to the parametric equation.

We start by splitting this single parametric equation into an equivalent system of two parametric equations,

$$\begin{pmatrix}1\\1\\1\end{pmatrix} t_1 + \begin{pmatrix}0\\0\\0\end{pmatrix} = \begin{pmatrix}A.a_2\\X.x\\I.o\end{pmatrix} \tag{10.4}$$

$$\begin{pmatrix}1\\1\\0\end{pmatrix} t_1 + \begin{pmatrix}1\\1\\1\end{pmatrix} t_2 + \begin{pmatrix}0\\0\\0\end{pmatrix} = \begin{pmatrix}A.a_1\\Y.y\\I.d\end{pmatrix} \tag{10.5}$$

Notice that Equation 10.4 is now in the join surface form. This will be one of the final join surfaces.

Now, let's examine Equation 10.5. What we are attempting to do is discover and *nest* join surfaces. Suppose that we decide that the join described by Equation 10.4 will be nested outermost. Then, if Equation 10.5 is placed within the body of this join, then the value of t_1 will be determined. In this case, the t_1 term of Equation 10.5 is constant, and the equation can be rewritten as,

$$\begin{pmatrix}1\\1\\1\end{pmatrix} t_2 + \underbrace{\begin{pmatrix}1\\1\\0\end{pmatrix} t_1 + \begin{pmatrix}0\\0\\0\end{pmatrix}}_{(*)} = \begin{pmatrix}A.a_1\\Y.y\\I.d\end{pmatrix} \tag{10.6}$$

Since the terms marked $(*)$ are constant inside the t_1 join, Equation 10.6 is now in the join surface form and is the second and last join surface.

10.3.3 The echelon form

We can take the equations of the two join surfaces from the previous example, Equations 10.4 and 10.6, and reconstitute them back into the following single system:

$$\underbrace{\begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}}_{\mathbf{H}'} \underbrace{\begin{pmatrix} t_1 \\ t_2 \end{pmatrix}}_{\mathbf{t}} \underset{+}{+} \underbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}}_{\bar{\mathbf{v}}'} \underset{=}{=} \underbrace{\begin{pmatrix} A.a_2 \\ X.x \\ I.o \\ A.a_1 \\ Y.y \\ I.d \end{pmatrix}}_{\mathbf{v}'}$$

The reason for doing this is to observe that $\mathbf{H}'$ is in column echelon form ([121]).

If $\mathbf{H}'$ is put into the column echelon form,

$$\left(\begin{array}{ccccc} 0 & 0 & 0 & & 0 \\ \boxed{\mathbf{h}'_1} & 0 & 0 & & 0 \\ \boxed{\mathbf{H}'_2} & \boxed{\mathbf{h}'_2} & 0 & \dots & 0 \\ \multicolumn{2}{c}{\boxed{\mathbf{H}'_3}} & \boxed{\mathbf{h}'_3} & & 0 \\ \vdots & & & \ddots & \\ \multicolumn{4}{c}{\boxed{\mathbf{H}'_n}} & \boxed{\mathbf{h}'_n} \end{array} \right) \tag{10.7}$$

then the parametric equation,

$$\mathbf{H}'\mathbf{t} + \bar{\mathbf{v}}' = \mathbf{v}' \tag{10.8}$$

has the form,

$$\begin{pmatrix} 0 & 0 & 0 & & 0 \\ \mathbf{h}'_1 & 0 & 0 & & 0 \\ \mathbf{H}'_2 \;\; \mathbf{h}'_2 & & 0 & \dots & 0 \\ \mathbf{H}'_3 & & \mathbf{h}'_3 & & 0 \\ & \vdots & & \ddots & \\ & \mathbf{H}'_n & & & \mathbf{h}'_n \end{pmatrix} \begin{pmatrix} t_1 \\ \vdots \\ t_n \end{pmatrix} + \begin{pmatrix} \bar{\mathbf{v}}'_0 \\ \bar{\mathbf{v}}'_1 \\ \vdots \\ \bar{\mathbf{v}}'_n \end{pmatrix} = \begin{pmatrix} \mathbf{v}'_0 \mathbf{v}'_1 \\ \vdots \\ \mathbf{v}'_n \end{pmatrix}$$

where each $\mathbf{v}'_k$, $\mathbf{h}'_k$ and $\bar{\mathbf{v}}'_k$ are vectors and each $\mathbf{H}'_k$ is a block matrix.

The first block row, when $k = 0$, occurs when one or more fields are accessed by constant values. The constraints on such fields will reduce to,

$$\bar{\mathbf{v}}'_0 = \mathbf{v}'_0$$

The remaining block rows where $k > 0$ correspond to the fields accessed by non-constant values. Each $\mathbf{v}'_k$ can be expressed as,

$$\mathbf{H}'_k \hat{\mathbf{t}}_k + \mathbf{h}'_k t_k + \bar{\mathbf{v}}'_k = \mathbf{v}'_k \tag{10.9}$$

where

$$\hat{\mathbf{t}}_k = \begin{pmatrix} t_1 \\ \vdots \\ t_{k-1} \end{pmatrix}$$

If we specify that the joins associated with the $\hat{\mathbf{t}}_k$ variables are nested outside of the t_k join, then $\hat{\mathbf{t}}_k$ is constant, and Equation 10.9 is in the join surface form. Thus, if the Equation 10.1 can be transformed into Equation 10.8, with $\mathbf{H}'$ in column echelon form, then the affine joins can be trivial discovered: there will be n such joins, and they will have the form of Equation 10.9.

10.4 Finding the echelon form

10.4.1 The standard technique

The standard algorithm for putting $\mathbf{H}$ into column echelon form is shown in Figure 10.1 ([121]). This algorithm computes a unimodular matrix, $\mathbf{Q}$, and column echelon matrix, $\mathbf{H}'$, such that $\mathbf{H}' = \mathbf{HQ}$. After running this this algorithm, we can easily read the join surfaces from $\mathbf{H}'$. This algorithm assumes that there is at least one non-zero in each row of $\mathbf{H}$, a restriction which will be relaxed in Section 10.5.3.

$\mathbf{H}', \mathbf{Q}$:= function $ColumnEchelonForm(\mathbf{H}_k)$
    $(m, n) := size(\mathbf{H}_k)$;
    $\mathbf{H}' := \mathbf{H}$; $\mathbf{Q} := \mathbf{I}$; $\langle i, j\rangle := \langle 1, 1\rangle$;
    while $i \leq m \wedge j \leq n$ do
        $\mathbf{Q}'$:= a set of column operations that make $\mathbf{H}(i, j)$
                non-zero and $\mathbf{H}(i, j+1:n)$ zero;
        $\mathbf{H}' := \mathbf{H}'\mathbf{Q}'$; $\mathbf{Q} := \mathbf{Q}\mathbf{Q}'$; $\langle i, j\rangle := \langle i+1, j+1\rangle$;
    end do
end function

Figure 10.1: Computing the column echelon form

The problem with this approach is that the join surfaces that we discover are dependent upon the order in which the index fields are assigned to rows in the original $\mathbf{H}$. Let's consider some examples,

Example 10.1 If $\mathbf{H}$ is taken from the parametric equation,

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} A.a_2 \\ X.x \\ I.o \\ A.a_1 \\ Y.y \\ I.d \end{pmatrix},$$

then $\mathbf{Q} = \mathbf{I}$ and $\mathbf{H}' = \mathbf{H}$, since $\mathbf{H}$ is already in echelon form. The join surfaces that are read from $\mathbf{H}'$ are on the fields $\langle A.a_2, X.x, I.o\rangle$ and $\langle A.a_1, Y.y, I.d\rangle$.

Example 10.2 If $\mathbf{H}$ is taken from the parametric equation,

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} I.o \\ I.d \\ A.a_1 \\ A.a_2 \\ X.x \\ Y.y \end{pmatrix},$$

then, again, $\mathbf{Q} = \mathbf{I}$ and $\mathbf{H}' = \mathbf{H}$, since $\mathbf{H}$ is already in echelon form. The join surfaces that are read from $\mathbf{H}'$ in this case are on the fields $\langle I.o\rangle$ and $\langle I.d, A.a_1, A.a_2, X.x, Y.y\rangle$.

To summarize, the same constraints can yield vastly different joins, based solely upon the order in which the index fields are assigned to rows.

10.4.2 Permuting the rows

This leads us to the following observation: if we apply a permutation, $\mathbf{P}$, to the rows of $\mathbf{H}$ before computing $\mathbf{Q}$, then we can control, the joins that will be discovered. In other words, let $\mathbf{H}' = (\mathbf{PH})\mathbf{Q}$. By considering different $\mathbf{P}$'s, we can explore many different nestings of the joins in the hopes of finding the most efficient one for the final sparse implementation. But what $\mathbf{P}$'s should we consider?

We calls rows in $\mathbf{H}$ that are multiples of one another *similar*. Since the standard algorithm in Figure 10.1 only performs column operations to $\mathbf{H}$, if several rows of $\mathbf{H}$ are similar, then they will also be similar in $\mathbf{H}'$. Thus, if a group of similar rows appear as the first rows of $\mathbf{H}$, then they will also be similar in $\mathbf{H}'$. Since the outermost join surface produced from $\mathbf{H}'$ is formed from the first row of $\mathbf{H}'$ and all subsequent rows with the same non-zero structure, this group of similar rows will form the outermost join surface.

Thus, one approach for finding a permutation, $\mathbf{P}$, is to

- identify the groups of similar rows of $\mathbf{H}$,

- decide which rows should form the outermost join, which group should form the second outermost join, and so on,
- form a permutation, $\mathbf{P}$, to move these groups of similar rows to their appropriate positions, and
- compute $\mathbf{H}' = (\mathbf{PH})\mathbf{Q}$ using the standard algorithm.

Example 10.3 In the parametric equation given in Example 10.2, there are three sets of similar rows, corresponding to the variables, $\langle I.o, A.a_2, X.x\rangle$, $\langle I.d\rangle$, and $\langle A.a_1, Y.y\rangle$. If we form a permutation to reorder the rows into the order

$$\langle I.o, A.a_2, X.x, I.d, A.a_1, Y.y\rangle$$

then the new $\mathbf{H}$ will be the same as in Example 10.1. So, the resulting joins will be on the fields $\langle A.a_2, X.x, I.o\rangle$ and $\langle A.a_1, Y.y, I.d\rangle$.

This example illustrates the problem with this approach. We started with three groups of similar rows, but we ended up with only two joins. Obviously, the set of similar rows in $\mathbf{H}$ do not exactly correspond to the final joins. What is happening?

10.4.3 Interleaving P and Q

If we examine the structure of Equation 10.7, which is used to form the final join surfaces, we can see what is happening. The kth join is formed from the rows that appear in the kth block of the echelon form. Some of these rows may be similar, in which case, the rows of $\mathbf{H}'_k$ will also be similar. However, this is not a necessary condition. That is, the rows of $\mathbf{H}'_k$ are not, in general, similar. Thus, non-similar rows of $\mathbf{H}$ can end up in the same join once the echelon form has been computed. In general, we cannot determine what rows will end up in which joins, a priori. So, computing $\mathbf{P}$ entirely before computing $\mathbf{Q}$ is of limited usefulness. A better approach is to interleave the computations of $\mathbf{P}$ and $\mathbf{Q}$.

The algorithm for interleaving the computation of $\mathbf{P}$ and $\mathbf{Q}$ should proceed in stages. At each stage, the algorithm takes $\mathbf{H}_k$, the portion of $\mathbf{H}$ that is not yet in echelon form and performs the following steps:

1. Compute the groups of similar rows in $\mathbf{H}_k$. Select one such group of rows as the next join surface to be scheduled.

$$\mathbf{H}_k = \begin{pmatrix} \vdots \\ \rule{3cm}{2pt} \\ \vdots \\ \rule{3cm}{2pt} \\ \vdots \end{pmatrix}$$

2. Construct a permutation, $\mathbf{P}_k$, that places these rows at the top of $\mathbf{H}_k$.

$$P_k H_k = \begin{pmatrix} \rule{3cm}{2pt} \\ \rule{3cm}{2pt} \\ \\ \vdots \\ \\ \end{pmatrix}$$

3. Construct a set of column operations, $\mathbf{Q}_k$, that annihilates all of entries in these rows, except for those in the first column.

$$\mathbf{P}_k \mathbf{H}_k \mathbf{Q}_k = \begin{pmatrix} x & 0 & \dots & 0 \\ x & 0 & \dots & 0 \\ \rule{2pt}{2cm} & & \mathbf{H}_{k+1} & \end{pmatrix} \tag{10.10}$$

The non-zero values $(x, x)^T$ will form $\mathbf{h}_k$.

4. Repeat this process with $\mathbf{H}_{k+1}$.

10.5 Non-deterministically finding P and Q

This basic idea for computing $\mathbf{P}$ and $\mathbf{Q}$ can be developed into a non-deterministic algorithm for computing $\mathbf{P}$ and $\mathbf{Q}$ in an interleaved manner. This algorithm is the culmination of the linear algebra framework for discovering join surfaces, and is the foundation for the join scheduling algorithm presented in Chapter 11. In this section, we will present two versions of this algorithm, one recursive and the other iterative. The recursive version is used to prove certain properties of correctness, while the iterative version is the one used in subsequent chapters.

10.5.1 The non-deterministic algorithm

Assume that $\mathbf{H}$ is the $m \times n$ matrix from the parametric equation shown in Equation 10.1. The algorithm in Figure 10.2 will use the four steps outlined in Section 10.4.3 to construct a $\mathbf{P}$ and $\mathbf{Q}$ such that $\mathbf{H}' = \mathbf{PHQ}$ is in column echelon form.

Finding Similar Rows. The function $FindSimilarRows(\mathbf{H}_k, i, j)$ divides the rows of $\mathbf{H}_k$ between i and m into *groups*, where each $group \in groups$ is a set of rows that are multiples of one another.

Computing the Permutation Transformation. The function $ComputePermutation(\mathbf{H}_k, group)$ computes a permutation $\mathbf{P}_k$ such that,

- All of the rows of $\mathbf{H}_k$ in *group*, will appear as the first rows of $\mathbf{P}_k\mathbf{H}_k$, and
- The rest of the rows of $\mathbf{H}_k$ will appear as the last rows of $\mathbf{P}_k\mathbf{H}_k$.

Computing the Annihilation Transformation. The algorithm for annihilating the non-zeros in $\mathbf{H}_k(1 : |group|, 2 : n)$ is shown in Figure 10.3. Since rows 1 through $|group|$ are multiples of one another, it suffices to perform column operations to zero row 1; the other rows in the group will be zeroed as well.

```
P, Q := function FindPandQ(H_k)
    (m, n) := size(H_k);
    if n = 0 then
        -- Base case of recursion.
        P := I; Q := I;
    else
        -- Step 1: find a group of similar rows.
        groups := FindSimilarRows(H_k);
        group := choose from groups;
        -- Step 2: find P_k.
        P_k := ComputePermutation(H_k, group);
        H_k := P_k H_k;
        -- Step 3: find Q_k.
        Q_k := ComputeAnnihilation(H_k);
        H_k := H_k Q_k;
        -- Step 4: continue with H_{k+1}.
        H_{k+1} := H(|group| + 1 : m, 2 : n);
        (P̂, Q̂) := FindPandQ(H_{k+1});
        -- Step 5: Construct P and Q and return.
        P := (I 0; 0 P̂) P_k; Q := Q_k (I 0; 0 Q̂);
    end if
end function
```

$$\mathbf{P} := \begin{pmatrix} \mathbf{I} & 0 \\ 0 & \hat{\mathbf{P}} \end{pmatrix} \mathbf{P}_k; \ \mathbf{Q} := \mathbf{Q}_k \begin{pmatrix} \mathbf{I} & 0 \\ 0 & \hat{\mathbf{Q}} \end{pmatrix};$$

Figure 10.2: Recursive Non-deterministic algorithm for computing $\mathbf{P}$ and $\mathbf{Q}$

$\mathbf{Q}_k :=$ function $ComputeAnnihilation(\mathbf{H}_k)$

  $(m, n) := size(\mathbf{H}_k)$;

  $\mathbf{Q}_k := \mathbf{I}$;

  for $j := 2$ to n do

    let $a = \mathbf{H}_k(1, 1)$, $b = \mathbf{H}_k(1, j)$;

    if $b = 0$ then continue j;

    -- *Annihilate* b

    $(d, u, v) := EGCD(a, b)$;

$$\tilde{\mathbf{Q}} := \begin{array}{cc} & \begin{array}{cccccccc} 1 & & & & j & & & \end{array} \\ \begin{array}{c} 1 \\ \\ \\ \\ j \\ \\ \\ \\ \end{array} & \begin{pmatrix} u & & & & -b/d & & & \\ & 1 & & & & & & \\ & & \ddots & & & & & \\ & & & 1 & & & & \\ v & & & & a/d & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{pmatrix} \end{array};$$

    $\mathbf{Q}_k := \mathbf{Q}_k \tilde{\mathbf{Q}}$; $\mathbf{H}_k := \mathbf{H}_k \tilde{\mathbf{Q}}$;

  end for

end function

Figure 10.3: Algorithm for computing $\mathbf{Q}_k$

Computing the Extended GCD. The $EGCD$, or Extended GCD, of two numbers a and b is usually defined as a function that returns values d, u, and v such that,

$$au + bv = d$$

where a, b, d, u, and v are all integers ([79]). However, we are interested in using the result to construct a unimodular transformation to annihilate b. In this case, $EGCD$ must compute d, u, and v such that

$$\begin{pmatrix} a & b \end{pmatrix} \underbrace{\begin{pmatrix} u & -b/d \\ v & a/d \end{pmatrix}}_{(*)} = \begin{pmatrix} d & 0 \end{pmatrix}$$

where $-b/d$ and a/d are integers. From this it can be shown that $(*)$ is unimodular. Such an algorithm for computing $EGCD(a, b)$ is given in [38].

10.5.2 Correctness

There are a few points that need to be argued about the correctness of this algorithm.

H has column full rank. If $\mathbf{H}$ does not have full column rank, then, for some k, the number of rows in $\mathbf{H}_k$ will be zero. The algorithm could be implemented in such a way as to gracefully handle this degenerate case, but that would be unnecessary: $\mathbf{H}$ will always have full column rank. Examining Equation 10.3, we can see that $\mathbf{I}_n$ is placed at the top of $\mathbf{H}$. Since $\mathbf{I}_n$ appears in the first rows of $\mathbf{H}$, $\mathbf{H}$ has full column rank.

P is a permutation transformation. $\mathbf{P}$ is a permutation transformation, since each of the $\mathbf{P}_k$'s is a permutation transformation, and $\mathbf{P}$ is the product of the $\mathbf{P}_k$'s.

Q is unimodular. $\mathbf{Q}$ is unimodular if each of the $\mathbf{Q}_k$'s is unimodular. $\mathbf{Q}_k$ is unimodular if its determinant is 1 or -1. $\mathbf{Q}_k$'s determinant is

$$\det \begin{pmatrix} u & & & & -b/d & & & \\ & 1 & & & & & & \\ & & \ddots & & & & & \\ & & & 1 & & & & \\ v & & & & a/d & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{pmatrix} =$$

$$\det \begin{pmatrix} u & -b/d \\ v & a/d \end{pmatrix} = \frac{au}{d} + \frac{bv}{d} = \frac{au+bv}{d} = 1$$

PHQ is in column echelon form.

Lemma 10.1 *ComputeAnnihilation, shown in Figure 10.3, zeros* $\mathbf{H}_k(1:|group|, 2:n)$

Proof *ComputeAnnihilation* zeros $\mathbf{H}_k(1, 2:n)$. Because rows 2 through $|group|$ of $\mathbf{H}_k$ are multiples of row 1, this will zero $\mathbf{H}_k(2:|group|, 2:n)$ as well.

Theorem 10.2 **PHQ** *is in column echelon form.*

Proof This can be shown by induction on the number of columns in **H**.

Base case: Assume that **H** has 0 columns. By definition, a zero-column **H** is in column echelon form.

Inductive case: Assume that the algorithm correctly puts matrices into column echelon form if they have n columns. We will show that a matrix with $n+1$ columns is put into column echelon form.

- *-- Step 1: find a group of similar rows.* The correctness of $FindSimilar-Rows$ is assumed.

- *-- Step 2: find* $\mathbf{P}_k$. The correctness of *ComputePermutation* is assumed.
- *-- Step 3: find* $\mathbf{Q}_k$. Lemma 10.1 tells us that *ComputeAnnihilation* zeros all but the first column of $\mathbf{H}_k(1 : |group|, 1 : m)$.
- *-- Step 4: continue with* $\mathbf{H}_{k+1}$. The induction assumption tells us that the recursive call to *FindPandQ* return $\hat{\mathbf{P}}$ and $\hat{\mathbf{Q}}$ such that $\mathbf{H}'_{k+1} = \hat{\mathbf{P}}\mathbf{H}_{k+1}\hat{\mathbf{Q}}$ is in column echelon form.
- *-- Step 5: Construct* $\mathbf{P}$ *and* $\mathbf{Q}$ *and return.* We have to show that $\mathbf{H}'_k = \mathbf{P}\mathbf{H}_k\mathbf{Q}$ is in echelon form.

$$\mathbf{H}'_k = \mathbf{P}\mathbf{H}_k\mathbf{Q} = \begin{pmatrix} \mathbf{I} & 0 \\ 0 & \hat{\mathbf{P}} \end{pmatrix} \mathbf{P}_k\mathbf{H}_k\mathbf{Q}_k \begin{pmatrix} \mathbf{I} & 0 \\ 0 & \hat{\mathbf{Q}} \end{pmatrix}$$

$$= \begin{pmatrix} \mathbf{I} & 0 \\ 0 & \hat{\mathbf{P}} \end{pmatrix} \begin{pmatrix} x & 0 & \dots & 0 \\ x & 0 & \dots & 0 \\ \Bigg| & & \mathbf{H}_{k+1} & \end{pmatrix} \begin{pmatrix} \mathbf{I} & 0 \\ 0 & \hat{\mathbf{Q}} \end{pmatrix}$$

$$= \begin{pmatrix} \mathbf{I} & 0 \\ 0 & \hat{\mathbf{P}} \end{pmatrix} \begin{pmatrix} x & 0 & \dots & 0 \\ x & 0 & \dots & 0 \\ \Bigg| & & \mathbf{H}_{k+1}\hat{\mathbf{Q}} & \end{pmatrix}$$

$$= \begin{pmatrix} x & 0 & \dots & 0 \\ x & 0 & \dots & 0 \\ \begin{pmatrix} \hat{\mathbf{P}} & \Bigg| \end{pmatrix} & & \hat{\mathbf{P}}\mathbf{H}_{k+1}\hat{\mathbf{Q}} & \end{pmatrix} = \begin{pmatrix} x & 0 & \dots & 0 \\ x & 0 & \dots & 0 \\ \begin{pmatrix} \hat{\mathbf{P}} & \Bigg| \end{pmatrix} & & \mathbf{H}'_{k+1} & \end{pmatrix}$$

Since $\mathbf{H}'_{k+1}$ is in column echelon form, $\mathbf{H}'_k$ is in column echelon form.

10.5.3 Constant indices

Up until this point, we have assumed that there were no zero rows in $\mathbf{H}$. Zero rows occur when the affine constraints in the query bind a field to a constant value. (e.g., $R.f = c$). Zero rows do not cause the algorithm for finding $\mathbf{P}$ and $\mathbf{Q}$ to fail. In fact, $\mathbf{H}' = \mathbf{PHQ}$ will be in echelon form.

However, $\mathbf{H}'$ is not guaranteed to have the structure shown in Equation 10.7. The reason is that the algorithm for finding $\mathbf{P}$ and $\mathbf{Q}$ makes no guarantee that the zero rows will be placed first in $\mathbf{H}'$. With only minor changes to the algorithm, the guarantee can be made. The gist of these changes are to ensure that the zero-rows and identified and handled as the first group of similar rows. No other modifications need to be made to account for constant indices.

10.5.4 An iterative version

It is easy to prove the correctness of the recursive formulation formulation of this algorithm, however, it would be cumbersome to use in subsequent discussions. In Figure 10.4, we present an iterative version of $FindPandQ$ that we will refer to in future discussions.

Note that this version writes the column echelon form of $\mathbf{H}$ back into $\mathbf{H}$, and does not return $\mathbf{P}$ and $\mathbf{Q}$. Also, notice that some of the subprocedures need to be modified for this formulation, but these changes are trivial enough that we do not bother presenting these modifications.

10.6 Running Example

Example 10.4 Dot-product

We can extract the following constraints from the dot-product query,

$$X.x = I.i$$
$$Y.y = I.i$$

We can express these affine constraints as,

```
procedure FindPandQ(H)
    (m, n) := size(H);
    i := 1; j := 1;
    -- (i, j) will denote the upper left corner of H_k.
    while j ≤ n do
        -- Step 1: find a group of similar rows.
        groups := FindSimilarRows(H, i);
        group := choose from groups;
        -- Step 2: find P_k.
        P_k := ComputePermutation(H, i, group);
        H := P_k H;
        -- Step 3: find Q_k.
        Q_k := ComputeAnnihilation(H, i);
        H := H Q_k;
        -- Step 4: continue with H_{k+1}.
        i := i + |group|;
    end do
end procedure
```

Figure 10.4: Iterative Non-deterministic algorithm for computing $\mathbf{P}$ and $\mathbf{Q}$

$$\underbrace{\begin{pmatrix}1\\1\\1\end{pmatrix}}_{\mathbf{H}} \quad \underbrace{\begin{pmatrix}t_1\end{pmatrix}}_{\mathbf{t}} \quad + \quad \underbrace{\begin{pmatrix}0\\0\\0\end{pmatrix}}_{\bar{\mathbf{v}}} \quad = \quad \underbrace{\begin{pmatrix}I.i\\X.x\\Y.y\end{pmatrix}}_{\mathbf{v}}$$

H is in echelon form, so no further transformations need to be performed.

Example 10.5 MVM

We can extract the following constraints from the MVM query,

$$\begin{aligned} A.a_1 &= I.i \\ A.a_2 &= I.j \\ Y.y &= I.i \\ X.x &= I.j \end{aligned}$$

We can express these affine constraints as,

$$\underbrace{\begin{pmatrix}1 & 0\\0 & 1\\1 & 0\\0 & 1\\1 & 0\\0 & 1\end{pmatrix}}_{\mathbf{H}} \quad \underbrace{\begin{pmatrix}t_1\\t_2\end{pmatrix}}_{\mathbf{t}} \quad + \quad \underbrace{\begin{pmatrix}0\\0\\0\\0\\0\\0\end{pmatrix}}_{\bar{\mathbf{v}}} \quad = \quad \underbrace{\begin{pmatrix}I.i\\I.j\\A.a_1\\A.a_2\\Y.y\\X.x\end{pmatrix}}_{\mathbf{v}}$$

H is in echelon form, so no further transformations are required. However, if we apply *FindPandQ* to **H**, then we will arrive at one of the following systems, which may yield a more efficient sparse implementation,

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} t'_1 \\ t'_2 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} I.i \\ A.a_1 \\ Y.y \\ I.j \\ A.a_2 \\ X.x \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} t''_1 \\ t''_2 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} I.j \\ A.a_2 \\ X.x \\ I.i \\ A.a_1 \\ Y.y \end{pmatrix}$$

The first of these transformed systems suggests accessing A by row, and the second suggests accessing A by column.

10.7 Related work

$\mathbf{H}$ in the initial parametric equation, $\mathbf{Ht} + \bar{\mathbf{v}} = \mathbf{v}$, is similar to the *Data Access Matrix* as described in [92] and [121]. Computing the echelon form of the data access matrix as part of program transformations to improve access patterns has previously been studied ([12], [13], [93]), but doing so for more than one dimension of matrix references appears to be novel.

The idea of using the echelon form of $\mathbf{H}$ to discover join surfaces in the query was first suggested by Kotlyar ([81]).

10.8 Summary

In this chapter, we have shown how the affine constraints from the query can be put into the parametric equation $\mathbf{Ht} + \bar{\mathbf{v}} = \mathbf{v}$. We have argued that the constraints of individual join surfaces can be expressed as the $\mathbf{h}t + \bar{\mathbf{v}} = \mathbf{v}$. We then showed that a set of nested join surfaces can be discovered after putting $\mathbf{H}$ into the echelon form shown in Equation 10.7. We have presented an algorithm for finding a $\mathbf{P}$ and $\mathbf{Q}$ in an interleaved manner, such that

$\mathbf{H}' = \mathbf{PHQ}$ is in the echelon form, and we have shown this algorithm to be correct.

Chapter 11

The Join Scheduler

At this point, we have shown how to discover hierarchies of indices, which give the feasible total orderings of the index and value fields of each relation, and have developed an interleaved algorithm for transforming $\mathbf{H}$ to $\mathbf{H}'$ in order to discover the nested join surfaces. Here, we combine the two techniques into a single algorithm. We call this algorithm, which transforms the query into a high-level plan, the *join scheduler*.

We will start this chapter by proving that the approach of scheduling nested single field joins is safe. Then, we introduce the notation that we will use for expressing high-level plans. Next, we will present a non-deterministic algorithm for performing join scheduling. After presenting the non-deterministic version of this algorithm, we will discuss heuristics that can be used to make it deterministic. We also will discuss how bounds for the final loop nests can be computed during this compilation phase.

11.1 Safety

In this section, we will show the correctness of scheduling queries as a sequence of nested single field joins that exploit the hierarchical structure of the sparse matrix storage formats.

Notation 11.1 (Disjoint Union)

The *disjoint union* of two relations is defined as $A \biguplus B = C$ iff

- $A \cup B = C$, and
- $A \neq B \Rightarrow A \cap B = \phi$.

The disjoint union of many relations is defined as, $\biguplus_k A_k = A$ iff

- $\cup_k A_k = A$, and
- $\forall_{k,l}, A_k \neq A_l \Rightarrow A_k \cap A_l = \phi$

We will use the disjoint union operator to mimic how a relation is "partitioned" by a loop. For instance, as a loop walks over the i field of a matrix in CRS format each of the compressed rows of the matrix is exposed in successive iterations of the the i loop. Thus, if A_i denotes the elements in the ith row of such a matrix, then the loop that enumerates the rows of A, partitions A as $\biguplus_i A_i = A$.

Note that for this purpose, the $\biguplus$ is not the same as the $\cup$ operator. The difference occurs when the same element occurs in several A_i's. If, by using a loop to partition a sparse matrix B, we were to visit the same entry of B more than one, than this would be an error. It would be an error because it would result in an iteration from the original dense specification being performed multiple times in the final sparse implementation. Thus, even though a loop might partition B as $\bigcup_k B_k = B$, it can be incorrect because of duplicate entries. If a loop partitions B as $\biguplus_k B_k = B$, then by definition duplicate entries cannot occur between different B_k's.

What follows are several results that culminate with Corollary 11.3, which states that our approach to partitioning queries for evaluation is correct. The proofs of these results presume the laws of the relational algebra operations that can be found in most database textbooks, [120] in particular.

Theorem 11.1 $\biguplus$ *distributes over* $\bowtie$.

$$B = \biguplus_k B_k \Rightarrow A \bowtie B = \biguplus_k (A \bowtie B_k)$$

Proof Assuming that $B = \biguplus_k B_k$,

- To show: $A \bowtie B = \bigcup_k (A \bowtie B_k)$

$$\begin{aligned} B = \biguplus_k B_k &\Rightarrow B = \cup_k B_k \\ &\Rightarrow A \bowtie B = A \bowtie \cup_k B_k = \cup_k (A \bowtie B_k) \end{aligned}$$

- To show: $A \bowtie B_k \neq A \bowtie B_l \Rightarrow (A \bowtie B_k) \cap (A \bowtie B_l) = \phi$

  $$\begin{array}{ll} A \bowtie B_k \neq A \bowtie B_l & \\ \quad \Rightarrow \pi\sigma(A \times B_k) \neq \pi\sigma(A \times B_l) & \\ \quad \Rightarrow A \times B_k \neq A \times B_l & (1) \\ \quad \Rightarrow A \neq A \vee B_k \neq B_l & \\ \quad \Rightarrow B_k \neq B_l & \\ \quad \Rightarrow B_k \cap B_l = \phi & \\ \quad \Rightarrow A \bowtie (B_k \cap B_l) = \phi & \\ \quad \Rightarrow \pi\sigma(A \times (B_k \cap B_l)) = \phi & \\ \quad \Rightarrow \pi\sigma((A \times B_k) \cap (A \times B_l)) = \phi & \\ \quad \Rightarrow \pi(\sigma(A \times B_k) \cap \sigma(A \times B_l)) = \phi & \\ \quad \Rightarrow \pi\sigma(A \times B_k) \cap \pi\sigma(A \times B_l) = \phi & (2) \\ \quad \Rightarrow (A \bowtie B_k) \cap (A \bowtie B_l) = \phi & \end{array}$$

  Notes:

  (1) From the definition of the natural inner join operator.

  (2) This is true only because the π removes the fields that are common to both relations.

- Since $A \bowtie B = \bigcup_k (A \bowtie B_k)$ and since $A \bowtie B_k \neq A \bowtie B_l \Rightarrow (A \bowtie B_k) \cap (A \bowtie B_l) = \phi$, it follows that $A \bowtie B = \biguplus_k (A \bowtie B_k)$. $\square$

Corollary 11.2 $A \bowtie B = \biguplus_{i,j}(A_i \bowtie B_j)$, *where* $A = \biguplus_i A_i$ *and* $B = \biguplus_j B_j$.

Corollary 11.3 *The query,*

```
for ⟨a, b⟩ ∈ A ⋈ B do
    ...
end do
```

can safely be executed as,

```
for ⟨A_i, B_j⟩ ⊂ A × B do
    for ⟨a, b⟩ ∈ A_i ⋈ B_j do
        ...
    end do
end do
```

where $A = \biguplus_i A_i$ *and* $B = \biguplus_j B_j$.

The final corollary tells us that it is safe to enumerate partitions A_i and B_j in an outer loop and perform their join in an inner loop. It suggests that all pairs of $\langle A_i, B_j \rangle$ must be enumerated; fortunately, this is not the case. If we can recognize that the join of some set of $\langle A_i, B_j \rangle$ pairs will be empty (i.e., $A_i \bowtie B_j = \phi$), then we can safely avoid enumerating those pairs.

The tuples stored in certain storage formats are accessed in hierarchical manner. That is, the indicing structure of these formats requires that the values of some fields be enumerated before the values of other fields can be accessed. Take the CRS format for instance: the value of row field i must be determined in order to access the non-zero entries that are stored within each of its compressed rows. These indexing structures form natural partitions of the tuples of the relations. In the case of CRS, the index on i forms a partition on the tuples, in the sense that, for a fixed value of i, all of the tuples with that value of i (i.e., that fall within row i) are easily accessed.

This condition can be expressed formally as, assuming that an indexing structure on field, f, partitions a sparse matrix A as $A = \biguplus_i A_i$, then $\forall A_i \subset A, \exists f', \{f'\} \pi_f A_i$.

It is precisely this sort of partitioning that we wish to exploit by generating nested single field joins. Corollary 11.4 proves the correctness of our approach.

Corollary 11.4 *Suppose that* f *is one of the common fields between* A *and* B*, and hence one of the fields participating in the join. Suppose also that* $A = \biguplus_i A_i$ *and* $B = \biguplus_j B_j$*, and*

- $\forall A_i \subset A, \exists f', \{f'\} \pi_f A_i$
- $\forall B_j \subset B, \exists f', \{f'\} \pi_f B_j$

then the following code will correctly compute $A \bowtie B$*,*

```
for f' ∈ (π_f A) ⋈ (π_f B) do
    for A_i ⊂ A where {f'} = π_f A_i do
        for B_i ⊂ B where {f'} = π_f B_j do
            for ⟨a, b⟩ ∈ A_i ⋈ B_j do
                ...
            end do
        end do
```

```
    end do
end do
```

Proof This code is based upon the loop nest shown in Corollary 11.3, except that it only enumerates those A_i's and B_j's for which $\pi_f A_i = \pi_f B_j$. This is safe, since if $\pi_f A_i \neq \pi_f B_j$, then $A_i \bowtie B_j = \phi$.

11.2 High-level plan

The join scheduler will take the following term from the query:

$$(I(\mathbf{i}, \mathbf{a}_1, \ldots, \mathbf{a}_p) \times A_1(\mathbf{a}_1, v_{a_1}) \times \cdots \times A_p(\mathbf{a}_p, v_{a_p}))$$

together with each of the affine constraints,

$$A_k.\mathbf{a}_k = \mathbf{F}_k \mathbf{i} + \mathbf{f}_k,$$

and derive a sequence of operations for evaluating this query, called a *high-level plan*. The tricky part about high-level plans is that they must allow the "global" portions of each query to be scheduled while leaving certain local portions unscheduled. After all, the join scheduler must leave enough freedom for join implementation to choose join strategies. Put another way, the high-level plan specifies what joins are to be done, and the order in which they are to be done, and the low-level plan specifies how each join will be implemented. We have already seen that hierarchies of indices can be used to impose an ordering on the fields of a relation without completely specifying a traversal. Hierarchies of traversals, together with information extracted from the linear system of constraints, will form the basis of the high-level plans.

11.2.1 Definition

A *high-level plan* is an integration of

- An ordering of the joins, as specified by a $\mathbf{P}$ and $\mathbf{Q}$, and
- A hierarchy of indices for each sparse matrix in the query.

into a single schedule. More specifically, a high-level plan is a sequence of stages, where each stage is one of,

- The kth join, which is comprised of the following:
  - The join surface of the join, which is the parametric equation described in Section 10.5,

$$\mathbf{v}'_k = \mathbf{H}'_k \hat{\mathbf{t}}_k + \mathbf{h}'_k t_k + \bar{\mathbf{v}}'_k,$$

    together with lb_t and ub_t, the bounds on t.
  - Zero or more join terms, each of which contains an index field that appear in $\mathbf{v}'_k$.
- A single term from one of the hierarchies of indices.

Additionally, a high-level plan has to satisfy certain sanity constraints,

Vertical Constraints.

- The order in which the join surfaces appear in the stages must be the same as the order that they appear in $\mathbf{H}'$. In other words, the kth join surface of $\mathbf{H}$ must appear in a stage prior to the $k+1$st join surface.
- The order in which terms from a hierarchy appear in the high-level plan must be the same as they appear in the hierarchy.

Horizontal Constraints.

- A join term may not appear in the high-level plan in a stage that is earlier than the join surface in which its index field appears.

The program representation produced by the join scheduler will have the form,

for $v \in \sigma_{QP}$ $\boxed{\text{High-level plan goes here}}$ do
    body
end do

It remains for the join implementer to take the high-level plan and the query predicate and to produce the low-level plan.

11.2.2 Examples

Example 11.1 Dot-product

Consider the query given for the dot-product running example. Suppose that $\mathbf{P}$ and $\mathbf{Q}$ are chosen such that the transformed parametric equation of the affine constraints is given by,

$$\overbrace{\underbrace{\begin{pmatrix} i \\ x.i \\ y.i \end{pmatrix}}_{\mathbf{v}'_1}}^{\mathbf{v}'} = \overbrace{\underbrace{\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}}_{\mathbf{h}'_1}}^{\mathbf{H}'} \overbrace{\underbrace{\begin{pmatrix} t_1 \end{pmatrix}}_{t_1}}^{\mathbf{t}} + \overbrace{\underbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}}_{\bar{\mathbf{v}}'_1}}^{\bar{\mathbf{v}}'}$$

Suppose also that the following hierarchies of indices are given for the sparse vectors, X and Y,

$$\begin{array}{rl} X: & \{X.i, X.ii\}, \{X.v\} \\ Y: & \{Y.i\}, \{Y.v\} \end{array}$$

Then the join scheduler might produce the following valid high-level plan for the query,

Stage	Linear	X	Y
1.	$\mathbf{v}'_1 = \mathbf{h}'_1 t_1 + \cdots$	$\{X.i, X.ii\}$	$\{Y.i\}$
2.		$\{X.v\}$	
3.			$\{Y.v\}$

Example 11.2 MVM

Consider the query given for the MVM running example. Suppose that $\mathbf{P}$ and $\mathbf{Q}$ are chosen such that the transformed parametric equation of the affine constraints is given by,

$$\overbrace{\begin{pmatrix} I.i \\ A.i \\ Y.i \\ I.j \\ A.j \\ X.j \end{pmatrix}}^{\mathbf{v}'} = \overbrace{\begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}}^{\mathbf{H}'} \overbrace{\begin{pmatrix} t_1 \\ t_2 \end{pmatrix}}^{\mathbf{t}} + \overbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}}^{\bar{\mathbf{v}}'}$$

with the equations for each of the two join surfaces being given by,

$$\overbrace{\begin{pmatrix} I.i \\ A.i \\ Y.i \end{pmatrix}}^{\mathbf{v}'_1} = \overbrace{\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}}^{\mathbf{h}'_1} t_1 + \overbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}}^{\bar{\mathbf{v}}'_1}$$

$$\overbrace{\begin{pmatrix} I.j \\ A.j \\ X.j \end{pmatrix}}^{\mathbf{v}'_2} = \overbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}}^{\mathbf{H}'_2} \overbrace{\begin{pmatrix} t_1 \end{pmatrix}}^{\hat{\mathbf{t}}'_2} + \overbrace{\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}}^{\mathbf{h}'_2} t_2 + \overbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}}^{\bar{\mathbf{v}}'_2}$$

Suppose also that the following hierarchies of indices are given for A, which is in CRS, X, which is a sparse vector, and Y, which is a dense vectors,

$$\begin{aligned} A: &\quad \{A.i\}, \{A.jj, A.j\}, \{A.v\} \\ X: &\quad \{X.i, X.ii\}, \{X.v\} \\ Y: &\quad \{Y.i\}, \{Y.v\} \end{aligned}$$

Then the join scheduler might produce the following valid high-level plan for the query,

Stage	Linear	A	X	Y
1.	$\mathbf{v}'_1 = \mathbf{h}'_1 t_1 + \cdots,$	$\{A.i\}$		$\{Y.i\}$
2.	$\mathbf{v}'_2 = \mathbf{h}'_1 t_2 + \cdots,$	$\{A.jj, A.j\}$	$\{X.j, X.jj\}$	
3.		$\{A.v\}$		
4.			$\{X.v\}$	
5.				$\{Y.v\}$

Example 11.3 The following is *not* a valid high-level plan for the previous example,

Stage	Linear	A	X	Y
1.				$\{Y.v\}$
2.	$\mathbf{v}'_1 = \mathbf{h}'_1 t_1 + \cdots$	$\{A.i\}$		$\{Y.i\}$
3.			$\{X.j, X.jj\}$	
4.	$\mathbf{v}'_2 = \mathbf{h}'_2 t_2 + \cdots$	$\{A.jj, A.j\}$		
5.		$\{A.v\}$		
6.			$\{X.v\}$	

A vertical constraint is violated in the Y column, because the term ordering, $\{Y.v\}$, $\{Y.i\}$ does not constitute a hierarchy of indices; there is no traversal that satisfies this order of the fields. A horizontal constraint is violated in row 3, because the index field $X.j$ appears in a stage before its corresponding join surface, which appears in row 4.

11.3 Non-deterministic algorithm for generating high-level plans

At this point, we have presented an algorithm for non-deterministically generating hierarchies of indices (Figure 9.2) and an algorithm for non-deterministically generating a $\mathbf{P}$ and $\mathbf{Q}$ (Figure 10.2). We need to develop a join scheduling algorithm that will fuse the results of these two algorithms and will produce a high-level plan. We could develop an algorithm for combining $\mathbf{P}$'s and $\mathbf{Q}$'s with hierarchies of indices to produce high-level plans, but we will instead develop an algorithm that simultaneously,

- Finds a $\mathbf{P}$ and $\mathbf{Q}$,

- Finds the hierarchies of indices, and
- Produces a high-level plan

There is an important reason for this approach. By developing a single algorithm instead of three, the non-determinism is concentrated in a single place, instead of being scattered over three algorithms. This will make it much easier to discuss heuristics for making the join scheduler deterministic.

This join scheduling algorithm will work by producing the high-level plan one stage at a time in a top-down manner. That is, the join scheduler starts with Stage 1 and finds the appropriate parts of $\mathbf{P}$ and $\mathbf{Q}$ and the appropriate terms hierarchies of indices that determine this stage. Then, when it is done with Stage 1, it does the same for Stage 2, and so on.

Before presenting this integrated algorithm, we need to define some terminology that it uses:

- A field of a sparse matrix is called *joinable* if it is an index field. Otherwise, it is called *unjoinable*.
- A term is joinable if it contains a joinable field. Otherwise, it is called *unjoinable*.
- A joinable field or term is said to be *determined* if its corresponding join surface has been placed in an earlier stage. In this case, the value of the field has already been established.

Example 11.4 In the CRS storage format, the i and j fields are joinable. The jj and v fields are unjoinable. In the following incomplete schedule, at Stage 3, both $X.j$ and $Y.i$ are determined since their correspond join surfaces have been scheduled in Stages 1 and 2.

Stage	Linear	A	X	Y
1.	$Join(\mathbf{v}'_1 = \mathbf{h}'_1 t'_1 + \cdots,$	$\{\{A.i\}$		$\})$
2.	$Join(\mathbf{v}'_2 = \mathbf{h}'_2 t'_2 + \cdots,$	$\{\{A.jj, A.j\}$		$\})$
3.	$\ldots$			

The non-deterministic join scheduling algorithm, *JoinScheduling*, is shown in Figure 11.1. The reader will be able to find the *FindHierarchy* algorithm and the *FindPandQ* algorithms embedded with in control constructs that sequence the stages and ensure that a valid high-level plan is constructed.

```
plan := function JoinScheduling(query)
    -- Initialize for computing the linear portion
    H, v, v̄ := FormParametricEquation(query);
    (m, n) := size(H);
    i := 1; k := 1;
    -- Initialize for computing the hierarchies of indices
    bound := φ;
    stage := 1; plan := ε;
    while AllFields(query) − bound ≠ φ do
        -- Schedule a single stage
        ready_terms :=
                  ⋃_{A ∈ SparseMatrices(query)} ComputeReadyTerms(A, bound);
        choose from
            -- Schedule a join
            [] k ≤ n →
                Schedule a single join using code from Figure 11.2
            -- Schedule a determined term
            [] term ∈ ready_terms ∧ IsJoinable?(term)
                    ∧IsDetermined?(term) →
                k := stage at which join for term was scheduled;

                step := Determined(k, term);
                newly_bound := term;
            -- Schedule an unjoinable term
            [] term ∈ ready_terms ∧ ¬IsJoinable?(term) →
                step := Unjoinable(term);
                newly_bound := term;
        end choose from
        plan := plan ++⟨step⟩;
        bound := bound ∪ newly_bound;
        stage := stage + 1;
    end do
end function
```

Figure 11.1: Non-deterministic algorithm for producing a high-level plan

```
-- Compute the linear portion of this stage ...
Execute steps 1-4 from Figure 10.4 here;

-- Extract the coefficient of the join surface ...
```
$\mathbf{H}_k := \mathbf{H}(i : i + |group| - 1, 1 : k - 1);$
$\mathbf{h}_k := \mathbf{H}(i : i + |group| - 1, k);$
$\mathbf{v}_k := \mathbf{v}(i : i + |group| - 1);\ \bar{\mathbf{v}}_k := \bar{\mathbf{v}}(i : i + |group| - 1);$

-- *Select the join terms for this stage* ...
$join\_terms := \phi;$
for $field \in VarsOf(\mathbf{v}_k)$ do
    if $\exists term \in ready\_terms$ s.t. $field \in term$ then
        $join\_terms := join\_terms \cup \{term\};$
    end if
end do

-- *Create this stage of the plan* ...
$step := Join(\mathbf{v}_k = \mathbf{H}_k\hat{\mathbf{t}}_k + \mathbf{h}_k t_k + \bar{\mathbf{v}}_k, bounds_{t_k}, join\_terms);$
$newly\_bound := \{field | field \in term \wedge term \in join\_terms\};$

Figure 11.2: Code for scheduling a single join

The *JoinScheduling* algorithm can deadlock if it reaches a stage where neither $k \leq n$ nor $ready\_terms \neq \phi$ is true. However, there will always be a set of non-deterministic choices that can be made to produce a valid high-level plan for a query, as long as a hierarchy of indices can be found for all of the sparse matrices. This will be possible as long as the access methods of each relation are sane. If the access methods are sane, then there exists at least one traversal for each sparse matrix, and thus at least one hierarchy of indices.

11.4 Our heuristic

There are three places where non-determinism occurs in the *JoinScheduling* algorithm,

- When choosing whether to perform a join at the current stage, or not,
- When selecting the group of similar rows when performing a join, and
- When selecting ready terms.

In this section, we will discuss the heuristic used by the current compiler to make these choices deterministic.

There is a key observation that have been built into this heuristic in several places: determined fields should be tested as soon as possible. There is a heuristic described in the database literature in which selections are performed as early as possible. The reason is to narrow the set of solutions produced as early in the evaluation of a query as possible. Our reasons are analogous: by hoisting the tests of determined fields as high as possible, we reduce the amount of useless computation that is performed when these tests fail.

Here is our heuristic.

1. If there are join variables that are constant (the corresponding rows of $\mathbf{H}$ are zero), then schedule the joins to which they correspond, with no join terms from any of the sparse matrices.

   Discussion: As we mentioned in Section 10.5.3, the zero-rows of $\mathbf{H}$ must be handled first, in order for the final $\mathbf{H}'$ to have the right shape. Another way to look at this is that constant fields are immediately scheduled, because they cannot participate in any of the joins that we are trying to discover.

2. If there are any ready lookup terms, they should be scheduled before ready enum terms.

    Discussion: The reason for scheduling lookup terms before enum terms is that, by placing them as early as possible, they will be invariant over as much of the high-level plan as possible. This is important because they usually correspond to either cheap dereferences or more expensive searches or lookups of data structures. In either case, it make sense to place these operations as high as possible.

    The lookup term to be scheduled is selected according to the following criteria,

    (a) If there is a lookup term whose field is determined, then schedule it.

        Discussion: As stated above, we aggressively try to place the tests for determined fields as high in the plan as possible.

    (b) If there is a lookup term whose field is joinable, then schedule the corresponding join.

        Discussion: We try to schedule joinable fields aggressively, because by placing them, the other fields participating in the join become determined. This will result in fewer joins and more searches being done in the inner stages of the plan.

    (c) Otherwise, randomly select a lookup term for scheduling.

3. Otherwise, if there are any ready enum terms, then schedule according to,

    (a) If there is a simple enum term whose index field is determined, then schedule it.

Discussion: A simple enum term is one that contains a single enum access method for the index field. If the field is determined, then a test must be performed to see if the value of the field is produced by the access method. Most frequently, the access method enumerates a range of integers, and a simple integer comparison test can be performed to see if the determined value of the field falls within the range.

(b) If there is either an indexing or searching enum term whose index field is determined, then schedule it.

Discussion: As stated above, we aggressively try to place the tests for determined fields as high in the plan as possible.

(c) If there is a join with more than one ready join term, then schedule it.

Discussion: A join with more than one ready join term represents a *non-trivial join*. We schedule these before other sorts of terms. The reason is that, by scheduling these joins as soon as possible, the constraints imposed by the join is hoisted as high in the loop nest as possible.

We pick a join based upon the following,

i. We select the join that contains the most number of strong references.

Discussion: Since each strong reference represents a necessary condition for the sparsity guard to be true, we place joins early in that plan the test as many of these necessary conditions as possible.

ii. If all joins contain the same number of strong references, a join is randomly selected.

(d) Otherwise, select an enum term for scheduling. If more than one enum term is ready, pick the enum term from an array before picking one from a vector.

Discussion: This is the "last chance" for the heuristic. At this point, only unjoinable enum terms and trivial joins are available. The heuristic must select one of the enums to schedule in order to make progress. By picking enum terms from arrays over vectors, the fields of the array are exposed before the fields of vectors. This will tend to result in searches of the vectors later in the plan, which is preferable, because searching the arrays in likely to be more expensive.

11.5 Computing loop bounds

Each join that is generated in the high-level plan has a corresponding join surface of the form,

$$\mathbf{v}'_k = \mathbf{H}'_k\hat{\mathbf{t}}_k + \mathbf{h}'_k t_k + \bar{\mathbf{v}}'_k$$

together with lb_t and ub_t, bounds on t, that are consistent with the original loop bounds. These bounds are easily computed using Fourier-Motzkin Elimination (FME) ([121]. FME will take

- The final linear system in echelon form, $\mathbf{H}'\mathbf{t} + \bar{\mathbf{v}}' = \mathbf{v}'$, and
- The original loop bounds as a system of linear inequalities, $\mathbf{Li} \leq \mathbf{b}$.

and form them into a system of linear inequalities in terms of the variables $\mathbf{t}$ and $\mathbf{v}'$,

$$\mathbf{L}'\begin{pmatrix}\mathbf{t}\\ \mathbf{v}'\end{pmatrix} \leq \mathbf{b}'$$

It will computes bounds of the t_k's by first projecting the $\mathbf{v}'$ variables out of this system to produce a system solely in terms of t_k's.

$$\mathbf{L}''\mathbf{t} \leq \mathbf{b}''$$

It will then project each t_k out of this system, in reverse order. A by-product of projecting t_k out of the system is its bounds in terms of $\hat{\mathbf{t}}_k$. Several systems are readily available for performing exactly these sorts of computations ([103], [54]).

Because **H**' is not known until $JoinScheduling$ is complete, FME cannot be performed until this point. In the current compiler, the join scheduler is implemented as a recursive procedure, which, as it recurses "down", performs join scheduling and, as it returns "up", performs FME.

11.6 Running examples

Example 11.5 Dot-product

We have already shown a high-level plan that might be constructed for our dot-product example in Example 11.1. Here is the sequence of stages that is actually produced by the join scheduler for this plan,

Stage	Linear	X	Y
1.	$Join(\mathbf{v}'_1 = \mathbf{h}'_1 t_1 + \cdots,$	$\{\{X.i, X.ii\},$	$\{Y.i\}\})$
2.	$Unjoinable($	$\{X.v\}$	$)$
3.	$Unjoinable($		$\{Y.v\})$

Example 11.6 MVM

We have already shown a high-level plan that might be constructed for our MVM example in Example 11.2. Here is the sequence of stages that is actually produced by the join scheduler for this plan,

Stage	Linear	A	X	Y
1.	$Join(\mathbf{v}'_1 = \mathbf{h}'_1 t_1 + \cdots,$	$\{\{A.i\},$		$\{Y.i\}\})$
2.	$Join(\mathbf{v}'_2 = \mathbf{h}'_2 t_2 + \cdots,$	$\{\{A.jj, A.j\},$	$\{X.j, X.jj\}$	$\})$
3.	$Unjoinable($	$\{A.v\}$		$)$
4.	$Unjoinable($		$\{X.v\}$	$)$
5.	$Unjoinable($			$\{Y.v\})$

11.7 Summary

In this chapter, we have showed how a query can be transformed into a high-level plan. We described a non-deterministic algorithm for performing this join scheduling, which combined the existing algorithms for producing hierarchies of indices and generating P and Q. We pointed out that the advantage of this interleaved algorithm was that it allowed all of the non-determinism to be isolated at one point in the algorithm. This single point of non-determinism makes it easier to discuss heuristics, which we did by presenting the one used in the current compiler. Finally, we discussed how the bounds for each of the t_k's could be computed.

Chapter 12

The Join Implementer

In this chapter we will discuss how a high-level plan is transformed into a low-level plan. We start this chapter by describing two algorithms that together constitutes the join implementer. The first algorithm is deterministic and is responsible for traversing the high-level plan and translating everything except for joins stages. The second algorithm is non-deterministic and is responsible for selecting the most appropriate implementation for each join stage. Finally, we will discuss what portion of this material has actually been implemented and what heuristics are used in the current implementation.

12.1 Preliminaries

In this section, we will elaborate on low-level plans and introduce notation that is used throughout this chapter.

12.1.1 Low-level plan

At this point in the sparse compilation process, we have transformed each query into a high-level plan, which consists of,

- A sequence of stages, which are one of,
  - A join, which is a join surface, bounds for each of the t_k's and any number of joinable terms,

$$Join(\mathbf{v}_k = \mathbf{H}_k\hat{\mathbf{t}}_k + \mathbf{h}_k t_k + \bar{\mathbf{v}}_k, bounds_{t_k}, \{T_1, \ldots, T_n\})$$

– A single determined term,

$$Determined(k, T)$$

– A single unjoinable term

$$Unjoinable(T)$$

- The query predicate from the original query, and
- The body of the query.

The join implementer is responsible for taking a high-level plan and producing a low-level plan for the instantiator. A low-level plan is essentially the final sparse implementation, except that abstract access methods are used instead of directly accessing the sparse matrix data structures. The instantiator will substitute the uses of these access methods with the code from the black-boxes in order to produce the final sparse implementation. In order to produce the low-level plan, we must

- Transform the sequence of stages in the high-level plan into a set of nested loops and conditionals expressed using access methods,
- Inserts code to test the bounds of each t_k, and

12.1.2 Notation

Notation 12.1 (Representing substitution maps)
Substitution maps are used to express bindings that are generated during join implementation. In particular, they are used to record

the expressions that are generated to produce the values of each of the relation fields.

- $[x \mapsto y]$ represents a substitution map that substitutes all occurrences of x with y.

$$[x \mapsto y]x \rightarrow y$$
$$[x \mapsto y]x' \rightarrow x' \text{when } x \neq x'$$

- $\mathcal{F}(c)$ means the substitution map, $\mathcal{F}$, applied to the code, c.
- $\mathcal{F}[x \mapsto y]$ represents the composition of the mapping function $\mathcal{F}$ to the substitution map $[x \mapsto y]$. More precisely, it means

$$(\mathcal{F}[x \mapsto y])(c) \equiv \text{if } x = c \text{ then } y \text{ else } \mathcal{F}(c)$$

- $bound?(\mathcal{F}(x))$ returns true iff there is a "binding" of x in $\mathcal{F}$. That is,

$$bound?(\mathcal{F}(x)) \equiv (\mathcal{F}(x) \neq x).$$

Notation 12.2 (Eliding input fields of access methods)
In order to generate code that invokes access methods, we need to generate the values of the input fields of each access method. If the expression,

$$A.infieldsOf(\texttt{name})$$

denotes the input fields of the access method, `name`, of the matrix, A, and $\mathcal{F}$ is the current substitution map from fields to expressions that will produce their value, then the expression,

$$A_{\texttt{name}}[\mathcal{F}(A.infieldsOf(\texttt{name}))]$$

will denotes the invocation of the access method with the correct input field values. However, this is tedious to read, so we will elide the input fields and use

$$A_{\texttt{name}}[\mathcal{F}(\diamond)]$$

to denote the same expression.

Computing fields and t_k's. The function $CalcField$ will return the value of field $A.f$, given the values of $\hat{\mathbf{t}}_k$ and t_k.

$x :=$ function $CalcField(k, A.f, \hat{\mathbf{t}}_k, t_k)$
  $x := [\mathbf{H}'_k]_{A.f}\hat{\mathbf{t}}_k + [\mathbf{h}'_k]_{A.f}t_k + [\bar{\mathbf{v}}'_k]_{A.f};$
end function

The function $Calc\_t_k$ will return the value of t_k, given the values of $\hat{\mathbf{t}}_k$ and x, the value of field $A.f$. $A.f$ must appear in the kth join of the current query.

$t_k :=$ function $Calc\_t_k(k, A.f, \hat{\mathbf{t}}_k, x)$

$$t_k := \frac{x - [\mathbf{H}'_k]_{A.f}\hat{\mathbf{t}}_k - [\bar{\mathbf{v}}'_k]_{A.f}}{[\mathbf{h}'_k]_{A.f}};$$

end function

In a sense, this is the inverse of $CalcField$.

Notation 12.3 (The value field of a sparse matrix)
In this presentation, we will assume that the name of the value field of all of the sparse matrices is v. So, if we want to refer to

the value field of A, we will use $A.v$. This sloppiness is much more readable than other notations that might be strictly correct, like $A.valueField()$.

Notation 12.4 (Delimiting generated code)
There are several places in the algorithms of this chapter in which the code is being generated. There is a need to visually distinguish the code of the algorithm from the code being generated by the algorithm. The code of the algorithm will,

Appear like this.

The code being generated by the algorithm will,

```
APPEAR LIKE THIS.
```

Furthermore, inside the code that is being generated by the algorithm, code may appear that is part of the execution of the algorithm. This represents an "escape" from the code being generated back to the algorithm.

Here is an extended example to illustrate the different levels at which this can occur,

This is code of the algorithm.
code :=

```
THIS IS CODE BEING GENERATED.
  This is an escape back to the algorithm.
  code' :=
      THIS IS CODE BEING GENERATED WITHIN THE
      ``ESCAPED'' CODE.
        and so on ...
```

If the reader is familiar with Lisp, they may recognize this as being similar to `quasiquote` and `unquote`.

12.2 The top-level algorithm

In this section, we will present the function, $JoinImplementation$, which performs all of the tasks of the join implementer, except selecting the implementation of each of the join stages.

12.2.1 The top-level

The function $JoinImplementation$ is shown in Figure 12.1. This function takes the original *query* and a high-level plan, *original_plan* generated by the join scheduler and returns the low-level plan, *code*.

```
code := function JoinImplementation(query, original_plan)

    -- J is shown in Figure 12.2.
    code := function J(plan, F)
        ...
    end function

    code := J(original_plan, [x ↦ x]);
end function
```

Figure 12.1: Top level of the join implementer

The function $\mathcal{J}$ is defined within $JoinImplementation$ and actually performs the work. $\mathcal{J}$ uses pattern matching on *plan* to determine what kind of stage is to be implemented. Then, depending upon what pattern clause is executed, different sorts of code are generated, and $\mathcal{J}$ is called recursively to generate the implementation for $plan'$, the remaining portion of the plan. $\mathcal{J}$ takes two parameters, *plan* and $\mathcal{F}$. *plan* is the portion of *original_plan* that remains to be processed. $\mathcal{F}$ is a substitution map that maps a field name, like $A.f$, to the low-level expression that has been created to generate its values. When calling $\mathcal{J}$ recursively to generate the low-level plan bindings for the fields that were generated by the current stage are added to $\mathcal{F}$. The top-level code of the function is shown in Figure 12.2.

$code :=$ function $\mathcal{J}(plan, \mathcal{F})$
    match $plan$ with
        pattern $Unjoinable(\{A.i\}^{?}_{\texttt{name}}) :: plan' \rightarrow$
            -- *Shown in Figure 12.3.*
        end pattern
        pattern $Determined(k, \{A.i\}^{?}_{\texttt{name}}) :: plan' \rightarrow$
            -- *Shown in Figure 12.4.*
        end pattern
        pattern $Determined(k, \{A.ii, A.i\}^{*}_{\texttt{enum\_ii}, \texttt{lookup\_i}}) :: plan' \rightarrow$
            -- *Shown in Figure 12.5.*
        end pattern
        pattern $Determined(k, \{A.ii, A.i\}^{*}_{\texttt{enum\_ii}, \texttt{lookup\_i}, \texttt{search\_i}}) :: plan' \rightarrow$
            -- *Shown in Figure 12.6.*
        end pattern
        pattern $Join(\mathbf{v}_k = \mathbf{H}_k\hat{\mathbf{t}}_k + \mathbf{h}_k t_k + \bar{\mathbf{v}}_k, bounds_{t_k}, \{T_1, \ldots, T_n\}) :: plan' \rightarrow$
            -- *Shown in Figure 12.7.*
        end pattern
        pattern $\langle\rangle \rightarrow$
            -- *Shown in Figure 12.8.*
        end pattern
    end match
end function

Figure 12.2: Code for $\mathcal{J}$

12.2.2 An unjoinable term

Perhaps the simplest kind of term is an unjoinable term. An unjoinable term binds the value of a field that does not appear in any join. The value field of a sparse matrix is a good example of an unjoinable term. The pattern for handling unjoinable terms is shown in Figure 12.3.

pattern $Unjoinable(term) :: plan' \rightarrow$
  let $\{A.f\}^{?}_{\texttt{name}} = term$;
  if $term = \{A.f\}^{1}_{\texttt{lookup\_f}}$ then
    -- *Case 1: $term$ is a lookup term*
    $code :=$

$$\boxed{\begin{array}{l}\texttt{IF } A_{\texttt{lookup\_f}}(\mathcal{F}(\diamond))\texttt{ THEN}\\ \qquad \boxed{\mathcal{J}(plan', \mathcal{F}[A.f \mapsto A_{\texttt{lookup\_f}}[\mathcal{F}(\diamond)]])}\\ \texttt{END IF}\end{array}}$$

;
  else if $term = \{A.f\}^{*}_{\texttt{enum\_f}}$ then
    -- *Case 2: $term$ is an enum term*
    $code :=$

$$\boxed{\begin{array}{l}\texttt{FOR } v \in A_{\texttt{enum\_f}}[\mathcal{F}(\diamond)]\texttt{ DO}\\ \qquad \boxed{\mathcal{J}(plan', \mathcal{F}[A.f \mapsto v])}\\ \texttt{END DO}\end{array}}$$

;
  end if
end pattern

Figure 12.3: An unjoinable term

There are only two cases to consider when generating the low-level plan for a unjoinable term,

- Case 1: $term$ is a lookup term. If $term$ is a lookup term, whose access method is the singleton method, `lookup_f`, then we generate code that first tests for the successful invocation of the access method. If the invocation succeeds, then $\mathcal{J}$ is called with $A.f$ bound to the result of invoking `lookup_f`. No code is generated for the case when the invocation does not succeed.

- Case 2: *term* is a enum term. If *term* is an enum term whose access method is the stream method, `enum_f`, then we generate a loop that enumerates the values in the stream returned by `enum_f`. In the body of this loop, we call $\mathcal{J}$ recursively to handle each element of the stream.

12.2.3 A 1-method determined term

The pattern for handling a 1-method determined term is shown in Figure 12.4. A determined term is one that contains a join field whose value has been determined by an enclosing stages of the high-level plan. The low-level plan needs to test that the value determined for the field actually exists within the specified relation. We handle this case in a manner similar to the unjoinable term case, except for one difference: in addition to looking up or enumerating the values of the access method, the generated code must test to determine that the value is equal to the determined value.

pattern $Determined(k, term) :: plan' \wedge \{A.i\}^{?}_{\texttt{name}} = term \rightarrow$
  if $term = \{A.i\}^{1}_{\texttt{lookup\_i}}$ then
    -- *Case 1: $term$ is a lookup term*
    $code :=$

$$\boxed{\begin{array}{l}\texttt{IF } A_{\texttt{lookup\_i}}(\mathcal{F}(\diamond)) \wedge A_{\texttt{lookup\_i}}[\mathcal{F}(\diamond)] = \mathcal{F}(A.i) \texttt{ THEN} \\ \qquad \boxed{\mathcal{J}(plan', \mathcal{F})} \\ \texttt{END IF}\end{array}}$$

;
  else if $term = \{A.i\}^{*}_{\texttt{enum\_i}}$ then
    -- *Case 2: $term$ is an enum term*
    $code :=$

$$\boxed{\begin{array}{l}\texttt{IF } \mathcal{F}(A.i) \in A_{\texttt{enum\_i}}[\mathcal{F}(\diamond)] \texttt{ THEN} \\ \qquad \boxed{\mathcal{J}(plan', \mathcal{F})} \\ \texttt{END IF}\end{array}}$$

;
  end if
end pattern

Figure 12.4: A 1-method determined term

12.2.4 A 2-method determined term

The pattern for handling a 2-method determined term is shown in Figure 12.5. By construction, a determined join term with 2-methods is an enum term. The stream method `enum_ii` is to be used to enumerate the values of the unjoinable field $A.ii$, and the singleton method `lookup_i`, is to be used to access the determined value of the index field, $A.i$. We have already presented the code for handling both so this clause simply pushes the appropriate stages back onto the plan and restarts $\mathcal{J}$.

$$
\begin{array}{l}
\textsf{pattern } \mathit{Determined}(k, \{A.ii, A.i\}^{*}_{\texttt{enum\_ii},\,\texttt{lookup\_i}}) :: \mathit{plan}' \rightarrow \\
\quad \mathit{new\_plan} := \mathit{Unjoinable}(\{A.ii\}^{*}_{\texttt{enum\_ii}}) :: \\
\qquad\qquad\qquad\qquad \mathit{Determined}(k, \{A.i\}^{1}_{\texttt{lookup\_i}}) :: \mathit{plan}'; \\
\quad \mathit{code} := \mathcal{J}(\mathit{new\_plan}, \mathcal{F}); \\
\textsf{end pattern}
\end{array}
$$

Figure 12.5: A 2-method determined term

12.2.5 A 3-method determined term

The pattern for handling a 3-method determined term is shown in Figure 12.6. The 3-method determined join term is similar to the 2-method case, except that a method, `search_i` is provided for binding the field $A.ii$, given the appropriate value of the field $A.i$. Since $A.i$ is already bound to the appropriate linear expression, we restart with a plan that binds $A.ii$ using this method.

$$
\begin{array}{l}
\textsf{pattern } \mathit{Determined}(k, \{A.ii, A.i\}^{*}_{\texttt{enum\_ii},\,\texttt{lookup\_i},\,\texttt{search\_i}}) :: \mathit{plan}' \rightarrow \\
\quad \mathit{new\_plan} := \mathit{Unjoinable}(\{A.ii\}^{1}_{\texttt{search\_ii}}) :: \mathit{plan}'; \\
\quad \mathit{code} := \mathcal{J}(\mathit{new\_plan}, \mathcal{F}); \\
\textsf{end pattern}
\end{array}
$$

Figure 12.6: A 3-method determined term

12.2.6 A join

The pattern for handling a join term is shown in Figure 12.7. This clause simply calls $SelectJoinImpl$, which is discussed in Section 12.3

pattern $Join(\mathbf{v}_k = \mathbf{H}_k\hat{\mathbf{t}}_k + \mathbf{h}_k t_k + \bar{\mathbf{v}}_k, bounds_{t_k}, \{T_1, \ldots, T_n\}) :: plan' \rightarrow$
    $code := SelectJoinImpl($
        $\mathbf{v}_k = \mathbf{H}_k\hat{\mathbf{t}}_k + \mathbf{h}_k t_k + \bar{\mathbf{v}}_k, bounds_{t_k},$
        $\{T_1, \ldots, T_n\}, plan');$
end pattern

Figure 12.7: A join

12.2.7 The body

Termination of the recursion occurs when *plan* is empty. At this point, the body of the high-level plan must be used to produce the body of the low-level plan. The code for this case is shown in Figure 12.8.

pattern $\langle\rangle \rightarrow$
    -- *Step 1. Replace placeholders.*
    let $\mathcal{E} = [x \mapsto x]$;
    for $A \in query.matrices$ do
        $\mathcal{E} := \mathcal{E}[BVAL(A.v) \mapsto \#t; LVAL(A.v) \mapsto \mathcal{F}(A.v);$
            $RVAL(A.v) \mapsto \mathcal{F}(A.v)];$
    end do
    -- *Step 2. Generate the body.*
    $code := \mathcal{E}(query.body);$
    end if
end pattern

Figure 12.8: The body

First, a substitution map, $\mathcal{E}$, is generated. $\mathcal{E}$ maps the sparse matrix placeholders (e.g., $BVAL(A.v)$, $LVAL(B.v)$, $RVAL(C.v)$) to expressions that have been generated by the various stages of the plan. The appropriate values can be found by applying $\mathcal{F}$ to the matrix's value field. Second, $\mathcal{E}$ is applied to the query's body to produce the low-level plan's body. In this way, each of the placeholders introduced during query formulation is replaced with the access method that will be used to perform the access in the final sparse implementation.

12.3 Selecting join implementations

The remaining task of join implementer is to select an implementation for each Join stage. Unfortunately, there are so many details and special cases to be handled when selecting a join implementation, that it does not make sense to present an entire algorithm for $SelectJoinImpl$ here. Instead, we will present a classification of join implementations and illustrate many of the join implementations that are commonly used in the relation database and sparse computation fields.

In order to make this presentation clearer, we will focus our discussion on a single join stage,

$$Join(\mathbf{v}_k = \mathbf{H}_k\hat{\mathbf{t}}_k + \mathbf{h}_k t_k + \bar{\mathbf{v}}_k, \qquad bounds_{t_k}, \{\{A.f\}^*_{\texttt{enum\_f}}, \{B.g\}^*_{\texttt{enum\_g}}\})$$

except where otherwise noted. Of course, these join implementations can easily be extended to other sorts of join stages.

The database literature does not present a consistent set of terminology for describing the many implementations of join operators. We will attempt to summarize these implementations using a classification of our own. This classification is certainly not exhaustive, but, as we will see, it includes many of the important join implementations from the database and sparse matrix literature.

There are three basic strategies for implementing joins

Enumerate and Select. The tuples from one relation are enumerated and, for each tuple from the first relation, the appropriate tuples from the second relation are retrieved.

Sort and Merge. First, the two relations are sorted by the join field. Second, the tuples from the two sorted relations are merged to produce the final results.

Blocking. Blocks of tuples are extracted from each relation and the join of these blocks is computed using some other join implementation.

12.3.1 Enumerate and Select

The basic form of this implementation is as follows,

```
-- Enumerate
for f_A ∈ A do
    if Calc_t_k(k, A.f, t̂_k, f_A) ∈ bounds_{t_k} then
        -- Select
        for g_B ∈ σ_{B.g=g_B} B do
            let F' = F[A.f ↦ f_A; B.g ↦ g_B];
            ...
        end do
    end if
end do
```

This is, one of the relations is selected as the target of the enumeration., In this case, we have arbitrarily selected A. For each tuple in A, we then select the relevant tuples from B. Enumerating over A is straightforward, but there are many choices for computing the selection on B.

Linear Search. The simplest method is to enumerate all tuples of B and test for matches with the current tuple of A. The code generated for such an implementation is shown in Figure 12.9. In the database literature, this implementation is called *selection on a product* ([120]) or *nested loop join* ([105]).

Efficient Search. We have assumed that the terms for both A and B are simple enum terms, but if the term for B was a searching enum term,

$$term_B = \{B.gg, B.g\}^*_{\texttt{enum\_gg},\,\texttt{lookup\_g},\,\texttt{search\_g}},$$

for $f_A \in A_{\texttt{enum\_f}}[\mathcal{F}(\diamond)]$ do
  $t_k := Calc\_t_k(k, A.f, \hat{\mathbf{t}}_k, f_A)$;
  if $t_k \in bounds_{t_k}$ then
    for $g_B \in B_{\texttt{enum\_g}}[\mathcal{F}(\diamond)]$ do
      if $t_k = Calc\_t_k(k, B.g, \hat{\mathbf{t}}_k, g_B)$ then
        let $\mathcal{F}' = \mathcal{F}[A.f \mapsto f_A; B.g \mapsto g_B]$;
        ...
      end if
    end do
  end if
end do

Figure 12.9: Nest loop join

then we can use its search method instead of the linear search. This is illustrated by the code in Figure 12.10. In the database literature, this implementation is called *join using an index* ([120]) or *index nested loop join* ([105]).

Creating an index. But, suppose that the join term for B does not have a search method. In this case, it is possible to generate code that will create an appropriate indexing structure, which has the effect of creating a search method that could be used by the join implementer. This technique is called *index creation* in the database literature ([120]).

As shown in Figure 12.11, we might create a hash table and then insert $\langle t_k, B.g\rangle$ entries in it, where t_k is the value of the parametric variable associated with $B.g$ Then, inside of the loop that enumerates the $A.f$'s, we can use the value of t_k generated from $A.f$ to search the hash table for the corresponding $B.g$'s.

Scatter/gather. A variant on this index creation method can be used when the plan contains the stages,

- $Join(\ldots, \ldots, \{\ldots, \{B\}^*_{\texttt{enum\_gg, lookup\_g}}\}))$ – an indexing join term for $B.gg$ and $B.g$,

for $f_A \in A_{\texttt{enum\_f}}[\mathcal{F}(\diamond)]$ do
  $t_k := Calc\_t_k(k, A.f, \hat{\mathbf{t}}_k, f_A)$;
  if $t_k \in bounds_{t_k}$ then
    let $\mathcal{F}' = \mathcal{F}[A.f \mapsto f_A;$
        $B.g \mapsto CalcField(k, B.g, \hat{\mathbf{t}}_k, t_k)]$;
    if $B_{\texttt{search\_g}}(\mathcal{F}'(\diamond))$ then
      let $\mathcal{F}'' = \mathcal{F}[B.gg = B_{\texttt{search\_g}}[\mathcal{F}'(\diamond)]]$;
      ...
    end if
  end if
end do

Figure 12.10: Index nested loop join

- $Unjoinable(\{B\}^1_{\texttt{lookup\_v}})$ – a lookup on the value field of B that has $B.gg$ as an input field, and
- no subsequent terms that use the value of the $B.gg$ field.

When this happens, we can place the values of $B.v$ directly into the hash table without storing $B.gg$ at all. This is most often done by allocating a dense vector V, which is indexed by $B.g$, and into which the values of $B.v$ are written. Before the join, the values of $B.v$ are copied into V, and, after the join, the values in V are written back into the appropriate locations of $B.v$. We call this a *scatter/gather* join, and its code is shown in Figure 12.12.

There are a few observations to be made about this code,

- V is assumed to be initialized to 0's when it is allocated. This is required, because indices that have not been initialized by the scatter operation may still be accessed. This happens when a value of $A.f$ does have a corresponding $B.g$ value in B.
- Notice that the amount of work required to set up and tear down V is proportional to the number of $B.gg$'s, and not the size of V. Obviously, this greatly improves the performance of the generated code.

```
-- Put the B.g's into the hashtbl_B, indexed by t_k.
initialize hashtbl_B;
for g_B ∈ B_enum_g[F(◇)] do
    t_k := Calc_t_k(k, B.g, t̂_k, g_B);
    if t_k ∈ bounds_{t_k} then
        add ⟨t_k, g_B⟩ to hashtbl_B;
    end if
end do
-- Enumerate over A.f's.
for f_A ∈ A_enum_f[F(◇)] do
    t_k := Calc_t_k(k, A.f, t̂_k, f_A);
    if t_k ∈ bounds_{t_k} then
        -- Search the hash table for B.g's.
        for g_B in hashtbl_B(t_k) do
            let F' = F[A.f ↦ f_A; B.g ↦ g_B];
            ...
        end do
    end if
end do
-- Tear the hash table down.
deallocate hashtbl_B;
```

Figure 12.11: Index creation

```
-- Scatter B
declare V : array 1 to n of real;
initialize set_{B.g};
for g_B ∈ B_enum_gg[F(◇)] do
    if B_lookup_g(F(◇)) ∧ B_lookup_v(F(◇)) then
        V[B_lookup_g[F(◇)]] := B_lookup_v[F(◇)];
        add B_lookup_g[F(◇)] to set_{B.g};
    end if
end do
-- Enumerate over A.f's.
for f_A ∈ A_enum_f[F(◇)] do
    t_k := Calc_t_k(k, A.f, t̂_k, f_A);
    if t_k ∈ bounds_{t_k} then
        g_B := CalcField(k, B.g, t̂_k, t_k);
        let F' = F[A.g ↦ f_A; B.g ↦ g_B; B.v ↦ V[g_B]];
        ...
    end if
end do
-- Gather B
for g_B ∈ B_enum_gg[F(◇)] do
    if B_lookup_g(F(◇)) ∧ B_lookup_v(F(◇)) then
        B_lookup_v[F(◇)] := V[B_lookup_g[F(◇)]];
    end if
end do
-- Tear the hash table down.
for i in set_{B.g} do
    V[i] := 0;
end do
deallocate set_{B.g}; deallocate V;
```

Figure 12.12: Scatter/gather join

- The gather operation can be eliminated if the value of $B.v$ is not updated.
- We have assumed that there are no duplicate $B.g$'s stored in B. That is, we have assumed that B's combining operator is "no duplicates." If B has some other combining operator, like "addition", then we can use the combining operator in the initialization of V as follows:

  if $B_{\texttt{lookup\_g}}[\mathcal{F}(\diamond)] \notin set_{B.g}$ then
  add $B_{\texttt{lookup\_g}}[\mathcal{F}(\diamond)]$ to $set_{B.g}$;
  $V[B_{\texttt{lookup\_g}}[\mathcal{F}(\diamond)]] := identity_{op}$;
  end if
  $V[B_{\texttt{lookup\_g}}[\mathcal{F}(\diamond)]] := V[B_{\texttt{lookup\_g}}[\mathcal{F}(\diamond)]]\ op\ B_{\texttt{lookup\_v}}[\mathcal{F}(\diamond)]$;

  However, in this case, we have no sensible way of writing the values of V back into $B.v$, so we only generate scatter code when $B.v$ is not updated.

12.3.2 Sort and Merge

The basic form sort and merge join implementation is,

$A' :=$ sort A on f;
$B' :=$ sort B on g;
for $\langle f_A, g_B\rangle \in merge(A', B')$ do
let $\mathcal{F}' = \mathcal{F}[A.f \mapsto f_A; B.g \mapsto g_B]$;
...
end do

In the database literature, this implementation is called *Sort-Join* ([120]) or *Sort-Merge join* ([105]).

Sorting. One can sort elements of each of $A.f$ and $B.g$ using, for instance, a heapsort or quicksort algorithm ([40]). Let's assume that heapsort is used and that the heaps A' and B' are created. Once the elements of $A.f$ and $B.g$ have been inserted into their appropriate data structures, assume that the elements can be extracted in sorted order using the access methods $A'_{\texttt{enum\_f}}[\,]$ and $B'_{\texttt{enum\_g}}[\,]$.

Of course, it may be that the elements of A or B are already sorted in the appropriate order. In this case, there is no need to construct the heap data structures and,

$$A'_{\texttt{enum\_f}}[\,] = A_{\texttt{enum\_f}}[\mathcal{F}(\diamond)] \qquad B'_{\texttt{enum\_g}}[\,] = B_{\texttt{enum\_g}}[\mathcal{F}(\diamond)].$$

The two-finger dot-product implementation shown in Figure 1.4 is an instance of this situation. For now, we will assume the existence of $Ordered?(term)$, a predicate that can be used to determine whether or not the values of the join field of $term$ are produced in *increasing* order. The $Ordered?$ predicate is discussed further in Section 12.5.1.

Merging. Once that appropriate methods $A'_{\texttt{enum\_f}}[\,]$ and $B'_{\texttt{enum\_g}}[\,]$ for producing $A.f$ and $B.g$ in increasing order have been constructed, it remains to merge these two streams of values.

First, it is necessary to determine whether or not the merge can be performed. Recall that a join is not being performed directly on the $A.f$ and $B.g$ fields. Rather, we are performing a join on the corresponding values of t_k. Since,

$$A.f = [\mathbf{H}_k]_{A.f}\hat{\mathbf{t}}_k + [\mathbf{h}_k]_{A.f}t_k + [\bar{\mathbf{v}}_k]_{A.f}$$

it follows that

- If $[\mathbf{h}_k]_{A.f} > 0$ and the stream of $A.f$ values is produced in increasing order, then the corresponding values of t_k for $A.f$ will be produced in increasing order.

- If $[\mathbf{h}_k]_{A.f} < 0$ and the stream of $A.f$ values is produced in increasing order then the corresponding values of t_k for $A.f$ will be produced in decreasing order.

- If $[\mathbf{h}_k]_{A.f} > 0$ and the stream of $A.f$ values is produced in decreasing order, then the corresponding values of t_k for $A.f$ will be produced in decreasing order.

- If $[\mathbf{h}_k]_{A.f} < 0$ and the stream of $A.f$ values is produced in decreasing order then the corresponding values of t_k for $A.f$ will be produced in increasing order.

Similar conditions holds for the t_k values for $B.g$.

A merge can only be performed on the two streams if they produce t_k's in the same order. Before choosing the Sort and Merge strategy, the join implementer must determine that the $A.f$'s and $B.g$'s can be provided in with an order such that the t_k's are enumerated with the same order. If the join implementer generating code to perform the sorting, then this can be assured; if the values of either of the relation are already ordered, then this strategy may not be feasible.

The other complication is that the multiple instances of a value of $A.f$ or $B.g$ may be encountered during the merge. In this case we must perform the cross-product of all such instances of $A.f$ with all such corresponding instances of $B.g$. This requires the use of a bag or similar data structure.

The code in Figure 12.13 illustrates the merge for the case when the t_k's are being produced in increasing order. The code for decreasing case is similar.

12.3.3 Blocking

The third strategy for implementing joins does not provide a complete implementation but a way to improve the performance of implementations obtained using the other two strategies. The basic form of this implementation is as follows,

for $A_i \subseteq A$ where $A = \biguplus_i A_i$ do
  for $B_j \subseteq B$ where $B = \biguplus_j B_i$ do
    for $\langle f_A, g_B \rangle \in (A_i \bowtie B_j)$ do
      ...
    end do
  end do
end do

That is, each of A and B are broken down into disjoint subsets, A_i and B_j respectively, and then the join of these two subrelations is computed. The

```
declare h_A : stream of A'_enum_f[ ]; h_A.init();
declare h_B : stream of B'_enum_f[ ]; h_B.init();
while h_A.valid() ∧ h_B.valid() do
    let f_A = h_A.deref(), t_kA = Calc_t_k(k, A.f, t̂_k, f_A);
    let g_B = h_B.deref(), t_kb = Calc_t_k(k, B.g, t̂_k, g_B);
    if t_kA = t_kB ∧ t_kB ∈ bounds_t_k then
        declare b_B : bag;
        while h_B.valid() ∧ h_B.deref() = g_B do
            add h_B.deref() to b_B;
            h_B.incr();
        end do
        while h_A.valid() ∧ h_A.deref() = f_A do
            for i_B := 1 to #b_B do
                let F' = F[A.f ↦ h_A.deref(); B.g ↦ b_B[i_B]];
                ...
            end do
            h_A.incr();
        end do
    else if t_kA > t_kB then
        h_B.incr();
    else
        h_A.incr();
    end if
end do
h_A.close(); h_B.close();
```

Figure 12.13: Merging two sorted relations

manner in which this nested join is implemented can be determined by calling *SelectJoinImpl* recursively.

Simple Blocking. Some of the join implementations that we have discussed so far, exhibit extremely poor data reuse. That is, these implementations do not make good use caches and other components of the memory hierarchy by reusing data soon after it has been accessed for the first time. The nested loop join implementation shown in Figure 12.9 is a good example of this poor reuse. Each g_B is examined many times–once for each value of f_A, in fact–but the number of intervening references between references to the same g_B can be quite large. This is because all of the tuples of B are touched before advancing to the next f_A. Since it is unlikely that all of the tuples of B will fit within the cache, each reference to g_B will result in a cache miss.

Consider a blocked version of this code as shown in Figure 12.14. If the sizes of each A_i and B_j are carefully chosen so that they both fit in cache together, then only the first reference to each g_B in the inner two loop will result in a cache miss. This particular join implementation is referred to as a *blocked nested loop join* in the database literature ([105]).

Rough hashing. Another instance where blocking is useful is in index creation. When we suggested using hash table for indices created on the fly, we ignored the fact that such a method can be very expensive in both time and space, if we require a single bucket for each key.

Suppose that we instead performed hashing twice. First, we put the tuples of A and B into two hash tables, $hashtbl_A$ and $hashtbl_B$ with a fixed number of buckets. Then, we perform a join on the corresponding buckets of $hashtbl_A$ and $hashtbl_B$, using the hashing join implementation described above and the smaller hash table $hashtbl'_B$. This is what is done in the code shown in Figure 12.15.

If this a blocked version of the original hashing join, then where are the two nested loops for corresponding to $A_i \subseteq A$ and $B_j \subseteq B$? In this case, generating two nested loops, such as,

for $w_A := 1$ to W do
  for $w_B := 1$ to W do
    for $\langle A.f, B.g \rangle \in (hashtbl_A[w_A] \bowtie hashtbl_B[w_B])$ do
      $\ldots$

```
for $A_i \subseteq A$ where $A = \biguplus_i A_i$ do
    for $B_j \subseteq B$ where $B = \biguplus_j B_i$ do
        for $f_A \in A_i$ do
            $t_k := Calc\_t_k(k, A.f, \hat{\mathbf{t}}_k, f_A)$;
            if $t_k \in bounds_{t_k}$ then
                for $g_B \in B_j$ do
                    if $t_k = Calc\_t_k(k, B.g, \hat{\mathbf{t}}_k, g_B)$ then
                        let $\mathcal{F}' = \mathcal{F}[A.f \mapsto f_A; B.g \mapsto g_B]$;
                        ...
                    end if
                end do
            end if
        end do
    end do
end do
```

Figure 12.14: Blocked Nested Loop

-- *Put the $A.f$'s into the $hashtbl_A$, indexed by t_k.*
initialize $hashtbl_A$ with buckets [1 ... W];
for $f_A \in A_{\texttt{enum\_f}}[\mathcal{F}(\diamond)]$ do
    $t_k := Calc\_t_k(k, A.f, \hat{\mathbf{t}}_k, f_A)$;
    if $t_k \in bounds_{t_k}$ then
        add $\langle t_k, f_A \rangle$ to $hashtbl_A$;
    end if
end do
-- *Put the $B.g$'s into the $hashtbl_B$, indexed by t_k.*
initialize $hashtbl_B$ with buckets [1 ... W];
for $g_B \in B_{\texttt{enum\_g}}[\mathcal{F}(\diamond)]$ do
    $t_k := Calc\_t_k(k, B.g, \hat{\mathbf{t}}_k, g_B)$;
    if $t_k \in bounds_{t_k}$ then
        add $\langle t_k, g_B \rangle$ to $hashtbl_B$;
    end if
end do
-- *Join each of the W buckets.*
for $w := 1$ to W do
    initialize $hashtbl'_B$;
    for $g_B \in hashtbl_B[w]$ do
        $t_k := Calc\_t_k(k, B.g, \hat{\mathbf{t}}_k, g_B)$;
        add $\langle t_k, g_B \rangle$ to $hashtbl'_B$;
    end do
    for $f_A \in hashtbl_A[w]$ do
        $t_k := Calc\_t_k(k, A.f, \hat{\mathbf{t}}_k, f_A)$;
        for g_B in $hashtbl'_B(t_k)$ do
            let $\mathcal{F}' = \mathcal{F}[A.f \mapsto f_A; B.g \mapsto g_B]$;
            ...
        end do
    end do
    deallocate $hashtbl'_B$;
end do
-- *Tear the hash tables down.*
deallocate $hashtbl_A$, $hashtbl_B$;

Figure 12.15: Rough hashing

```
        end do
    end do
end do
```

would be unnecessary, since we know that the join is empty except when $w_A = w_B$.

This join implementation is referred to as the *blocked hash join* in the database literature ([105]) and offers several advantages over the unblocked hash join,

- The blocked version exhibits better reuse than the unblocked version and should performance benefits from better use of the memory hierarchy.
- Because a smaller number of tuples are being hashed in the inner loop, the overhead of inserting the tuples should be lower because fewer collisions are likely to occur.

12.4 Heuristics/Policies

In the previous section, we discussed many ways of implementing joins, but we did not discuss under what circumstances each technique should be applied. Another way to say this is, we have presented the join implementations, but what about the join implementer's policies? At the moment, we have not developed a complete heuristic that can be used to build a robust join implementer. We plan to do so in the future. In the mean time, we do have some insights into some of the issues that would have to be addressed in developing such a heuristic. We present those insights here.

We will limit discussion to just three particular join implementations,

- the selection on a product,
- the sort and merge join in which both relations are already sorted, and
- the scatter/gather version of reindexing.

Which of these implementation should be used in which circumstances? Clearly, selection on a product should be the implementation of last resort, because of its quadratic behavior. But what of the other two?

The scatter/gather implementation is not appropriate in all circumstances. In particular, when generating a sparse implementation for a node program for a parallel machine, it not appropriate to generate an accumulator of size n, where n is proportional to the problem size. The reason is that, if every node of a parallel processor allocates a size n vector, the sparse implementation will not scalar well as the problem size or number of processors is increased. Of course, data structures that take space proportional to the number elements stored (e.g., a hash table) could still be used.

Another consideration is which of the two implementations would be more efficient As we discussed in Section 1.6.3 and illustrated in Figure 1.5, that depends upon the circumstances in which the join appears. If we examine a comparison of the performance of a two-finger join, a scatter/gather join without amortization, and a scatter/gather join with amortization, as shown in Figure 1.5, it is clear that amortization makes a significant difference. The bottom line seems to be this: if the cost of the reindexing operations can be amortized, then the scatter/gather join is to be prefered; otherwise, the two-finger join is to be prefered.

12.5 Unresolved issues

There are some unresolved issues that we have so far conveniently swept under the rug, but which now must be dragged out into the light of day.

12.5.1 Ordering predicate

In Section 12.3.2, it was stated that a $Ordered?(\ldots)$ predicate, or one like it, is required in order to determine whether or not join terms produce their values in increasing or decreasing order. The present version of the black-box protocol, as described in Appendix C does not provide the information needed to implement this predicate. It would be relatively minor to add, and we plan to do so in the future.

12.5.2 The current heuristic

The primary reason why we have not yet developed a complete heuristic for join implementation is because we have not found it necessary for the code that we have focused on. Those codes, which include MVM and MMM, were simple enough that the following heuristic proved sufficient for obtaining efficient sparse implementations,

1. If a join stage contains any indexing terms, then place them outermost using selection on a product. We do not attempt to generate either two-finger or scatter/gather joins in the current implementation.

2. If a join stage contains any searching terms, place them inside the indexing terms.

3. If a join stage contains any enum terms, then place them inside the search terms. In practice, we have found that the code generated for determined enum terms is almost always a simple for loop containing a conditional, as in the following

   for $f \in A_{\texttt{enum\_i}}[\mathcal{F}(\diamond)]$ do
       if $f = i$ then
           ...
       end if
   end if

   $\Downarrow$ instantiation

   for $f := lb$ to ub do
       if $f = i$ then
           ...
       end if
   end if

   This case can be recognized and replaced with the following range test,

   if $lb \leq i \leq ub$ then
       $f := i$;
       ...
   end if

> In fact, in the current compiler, we use FME to try and eliminate even this test.

This heuristic is clearly inadequate for a general purpose compiler. We plan to develop a more robust heuristic in the future.

12.6 Running Example

Example 12.1 Dot-product

Assume that the join implementer chooses to implement the join in the dot-product high-level plan in Example 11.5 by enumerating the indices in X and then using the `search_i` method to search into Y. Then, the join implementer might produce the following low-level plan,

$sum := 0;$
for $ii_X \in X_{\texttt{enum\_ii}}[\,]$ do
  $i := X_{\texttt{lookup\_i}}[ii_X];$
  if $Y_{\texttt{search\_i}}(i)$ then
    $ii_Y := Y_{\texttt{search\_i}}[i];$
    $sum := sum + X_{\texttt{lookup\_v}}[ii_X] * Y_{\texttt{lookup\_i}}[ii_Y];$
  end if
end do

Example 12.2 MVM

Assume that the join implementer recognizes that, since Y is dense, the outermost join of the MVM high-level plan shown in Example 11.6 can be implemented by just enumerating the i indices of A. The current implementation of the join implementer is able to do this using the techniques that will be discussed in Section 14.3. Let's also assume that it chooses to scatter X in order to perform the innermost join. Then, the join implementer might produce the following low-level plan,

-- *Scatter* X *to* T
for $ii_X \in X_{\texttt{enum\_ii}}[\,]$ do
  $T[X_{\texttt{lookup\_i}}[ii_X]] := X_{\texttt{lookup\_v}}[ii_X];$
end do
for $i_A \in A_{\texttt{enum\_i}}[\,]$ do

```
    for $jj_A \in A_{\texttt{enum\_jj}}[i_A]$ do
        $y[i_A] := y[i_A] + A_{\texttt{lookup\_v}}[jj_A] * T[A_{\texttt{lookup\_j}}[jj_A]]$;
    end do
end do
```

12.7 Summary

In this chapter, we have provided a complete set of mechanisms for transforming high-level plans into low-level plans, which, for reasons that should be obvious now, we have called the join implementer. We have stated that, to date, we do not have a general heuristic for driving all of these techniques, but we did provide some insights that such a heuristic would have to consider, and we have presented the minimal heuristic that we use in the current implementation. The methods discussed in this chapter are intended for queries constructed with only $\bowtie$ operators. More general queries will be discussed in Chapter 15.

Chapter 13

Instantiation

In this chapter, we will discuss the process of transforming the low-level plan into the final object code. We do this transformation in two stages. First, we replace the uses of access methods, like $A_{\texttt{search\_i}}(\dots)$ and $B_{\texttt{enum\_jj}}[\dots]$, with the appropriate implementations, as obtained from the appropriate black-boxes. This process is called *instantiation*. Second, we generate the final object code from the instantiated code.

13.1 The sparse implementation

The low-level plan, which is the output of the join implementer, is the code in which,

- The query and all of its joins have been scheduled to loops and conditionals, but
- All accesses to sparse matrices are expressed in terms of the abstract access methods.

There are two sorts of access methods,

- Singleton access methods, which can appear either as
  - a boolean test of the existence of a particular value, or $A_{\texttt{name}}(\dots)$,
  - a reference to a particular value, $A_{\texttt{name}}[\dots]$.
- Stream access methods, which can appear as

- the initializer of a stream declaration, as in,

  declare h : stream of $A_{\texttt{enum\_ii}}[\ldots]$;

- the initializer for a loop that enumerates the stream, as in,

  for $v \in A_{\texttt{enum\_ii}}[\ldots]$ do
  $\ldots$
  end do

- the target of a set inclusion test, as in,

  if $v \in A_{\texttt{enum\_ii}}[\ldots]$ then
  $\ldots$
  end do

If each of the black-boxes were to provide a library containing implementations of each of these access methods, then the low-level plan could serve as the final sparse implementation. All that would be required to make it executable is to translate it to C or FORTRAN, compile it, and link it the routines provided by the black-boxes.

However, this approach will be an impediment to obtaining the best performance for the sparse implementation. As we will see in Chapter 14, it is imperative that the sparse compiler be able to replace some of the access methods with their implementation in order to perform conventional dense optimizations. If the implementation of the access methods is not provided until link-time, this will not be possible.

Instead of the black-boxes providing the run-time implementations of the access methods, they will instead provide a means for generating these implementations at compile-time. The black-box protocol provides an interface between the compile and the black-boxes that allows program representations to be between the two in order for this generation to take place. The instantiator replaces all access methods uses with the concrete implementation obtained from the corresponding storage format black-boxes.

The resulting code will be at roughly the same semantic level as C or FORTRAN. That is, it will no longer contain such high-level concepts as queries or joins, or even abstract accesses to sparse matrix formats. Instead, the code will contain all of the loops, conditionals, or array references calls of the final implementation. At this point, conventional techniques can be used to generate the final object code.

We will first discuss how stream access methods are instantiated. Then, we will discuss how singleton access methods are instantiated. Finally, we will discuss some issues of compiling the final sparse implementation into object code.

13.2 Instantiating stream accesses

This section describes the portion of the black-box protocol responsible for instantiating stream access methods. A detailed description of this portion of the protocol can be found in Appendix C.1.5. In particular, the reader should examine the declaration and use of `am_stream_type`.

13.2.1 Instantiation functions for stream methods

A stream access method of a sparse matrix storage format provides, via the black-box protocol, a single function for transforming an access method into its implementation. If A is a sparse matrix, and `name` is an access method of A, then $A.instOf$(`name`) will denote the function for performing this instantiation, or the *instantiation function*. An instantiation function for a stream access method has the following type signature,

$A.instOf$(`name`):
function taking
args: list of $\mathcal{E}$,
and returning
decl_ids: list of $\mathcal{I}$,
init_st: $\mathcal{S}$,
valid_ex: $\mathcal{S}$,
deref_ex: $\mathcal{E}$,
incr_st: $\mathcal{E}$,
close_st: $\mathcal{S}$

In this signature, $\mathcal{I}$, $\mathcal{E}$ and $\mathcal{S}$ are the types of identifier, expression, and statement Abstract Syntax Trees (AST's), respectively, used within the compiler. It is important to notice that an instantiation function is a transformation on AST's within the compiler. An instantiation function does not perform the access; it generates code at compile-time that will perform the access at run-time.

Each of the arguments and results has the following meaning,

- Argument *args* is a list of the expression AST's that are the arguments to the access method. That is, if the access method appearing in the low-level plan is $A_{\mathtt{name}}[x, y, z]$, then a list of the AST's for the expressions x, y, and z will be passed as the value of *args*.
- Result *decls_ids* is a list of variable declarations that are required to implement the stream handle produced by the access method.
- Result *init_st* is a statement AST that will initialize the stream handler.
- Result *valid_ex* is an expression AST that will return $\#t$ iff the stream is non-empty.
- Result *deref_ex* is an expression AST that will return the value at the current position in the stream.
- Result *incr_st* is a statement AST that will advance the handler to the next position in the stream.
- Result *close_st* is a statement AST that will clear the stream handler and tear down any data structures created by *init_st*.

13.2.2 Generating general stream accesses

Let us suppose that the following fragment of code appeared in the low-level plan,

> declare h : stream of $A_{\mathtt{enum\_i}}[\;]$;
> $h.init()$;
> while $\ldots h.valid() \ldots$ do
> $\ldots$
> $\ldots h.deref() \ldots$
> $\ldots$
> $h_B.incr()$;
> $\ldots$

```
end do
h.close();
```

And, let us also suppose that the implementation of this stream simply enumerated the values from 1 to n. Then, the instantiation function might return the following results,

- $decl\_ids$: declare i : int;
- $init\_st$: $i := 1$;
- $valid\_ex$: $i < n$
- $deref\_ex$: i
- $incr\_st$: $i := i + 1$;
- $close\_st$: nop

Bear in mind that all of these results are AST's. Once these AST's have been substituted into the low-level plan, the final sparse implementation will be,

```
declare i : int;
i := i';
while i < n do
    ...
   ...i...
    ...
   i := i + 1;
    ...
end do
nop;
```

In order to replace stream access methods with their implementation, the instantiator must contain functionality similar to the following:

$code' :=$ function $Instantiate(code, \mathcal{F})$

```
⋮

-- Handle "declare h : stream of A_enum_i[ ]; S;"
(decl_ids, init_st, valid_ex, deref_ex, incr_st, close_st) :=
        A.instOf(enum_i)();
code' := decl_ids ++
        ⟨Instantiate(S, F[
            h.init() ↦ init_st;
            h.valid() ↦ valid_ex;
            h.deref() ↦ deref_ex;
            h.incr() ↦ incr_st;
            h.close() ↦ close_st;
        ])⟩;

⋮

end function
```

13.2.3 Generating loops over stream accesses

The loop short-hand for stream access methods, as in the following,

```
for v ∈ A_enum_f[...] do
    body;
end do
```

can easily be instantiated using the existing method by first transforming the above code to the following,

```
declare h : stream of A_enum_f[...];
h.init();
while h.valid() do
    let v = h.deref()
    body;
    h_B.incr();
end do
```

$h.close()$;

13.2.4 Generating membership tests of stream accesses

The membership test for stream access methods, as in the following,

if $f' \in A_{\texttt{enum\_f}}[\ldots]$ then
    body;
end if

can easily be instantiated using the existing method, by first transforming the above code to the following,

declare h : stream of $A_{\texttt{enum\_f}}[\ldots]$;
$found := \#f$;
$h.init()$;
while $h.valid() \wedge \neg found$ do
    if $f' = h.deref()$ then
        body;
        $found := \#t$;
    else
        $h_B.incr()$;
    end if
end do
$h.close()$;

13.3 Instantiating singleton accesses

This section describes the portion of the black-box protocol responsible for instantiating singleton access methods. A detailed description of this portion of the protocol can be found in Appendix C.1.4. In particular, the reader should examine the declaration and use of `am_search_type`.

13.3.1 Instantiation functions for singleton methods

An instantiation function for a singleton access method has the following type signature,

$A.instOf(\texttt{name})$:
function taking

$args$:	list of $\mathcal{E}$,
$found\_f$:	function taking $\mathcal{E}$ and returning $\mathcal{S}$,
not_found_f :	function taking no arguments and returning $\mathcal{S}$

and returning

$code$:	$\mathcal{S}$

Each of the arguments and results has the following meaning,

- Argument $args$ is a list of the expression AST's that are arguments to the access method. That is, if the access method in the low-level plan is $A_{\texttt{name}}[x, y, z]$, then a list of the AST's for the expressions x, y, and z will be the value of $args$.
- Argument $found\_f$ is a function that takes an expression AST and returns a statement AST. This function is provided by the compiler and is called by the black-box to generate code for the case when the value being accessed is found in the sparse matrix. The argument to $found\_f$ is an expression AST for the reference generated by the access method.
- Argument not_found_f is a function that has no arguments and returns a statement AST. This function is provided by the compiler and is called by the black-box to generate code for the case when the value being accessed is not found.
- Result $code$ is the final code returned by the instantiation function.

The meaning of the $found\_f$ and $\not{f}ound\_f$ arguments may not be clear, so we will discuss them further.

13.3.2 Why the protocol is higher-order

Consider a fragment of a low-level plan that might appear as input to the instantiator.

```
if v_search_i(i') then
    S1: ... v_search_i[i'] ...
else
    S2: ...
end if
```

Suppose that v's storage format is a sparse vector, and that the `search_i` access method is implemented using the following code, which performs binary search:

```
i := -- T1: The value to search for goes here;
low := 1;
hig := #i's;
while (low < hig) do
    mid := (low + hig)/2;
    if i ≤ vind[mid] then
        hig := mid;
    else
        low := mid + 1;
    end if
end do
if i = vind[low] then
    -- T2: Code for when i is found goes here.
else
    -- T3: Code for when i is not found goes here.
end if
```

In order to instantiate the access methods in the low-level code, the sparse vector black-box must generate a copy of the binary search code, substituting,

- i' for -- *T1*,
- S1 for -- *T2*,

- S2 for -- *T3*,

and within S1 substituting low, the result of the search, for all occurrences of $v_{\texttt{search\_i}}[i']$. The final code will look like,

```
i := i';
low := 1;
hig := #i's;
while (low < hig) do
    mid := (low + hig)/2;
    if i ≤ vind[mid] then
        hig := mid;
    else
        low := mid + 1;
    end if
end do
if i = vind[low] then
    S1: ... low ...
else
    S2: ...
end if
```

One obvious approach to producing this code would be for the instantiation function to perform exactly the steps described above,

1. Generate a copy of the binary search code,
2. Substitute the index value, the "found" code, and the "not found" code into the binary search, and
3. Substitute low for $v_{\texttt{search\_i}}[i']$ in the "found" code.

While very simple to describe, this approach is extremely tedious and inefficient to implement. If several access methods occur within an AST, then repeated substitutions will have to be performed over the AST. If each substitution is performed entirely within the instantiation function that requires it, then these multiple substitutions will require multiple traversals of the AST. This is very inefficient. A different approach would be for the instantiation function to return a list of substitution that need to be performed, but this unnecessarily complicates the black-box protocol.

A better approach is for the compiler to pass functions to generate the "found" and "not found" codes to the instantiation function, instead of passing the codes directly. Here is what occurs:

1. The compiler invokes the instantiation function, passing the access method arguments and functions for generating the "found" and "not found" codes.

2. The instantiation function generates a copy of the binary search code. It invokes the $found\_f$ function to generate the code for the "found" case, and the not_found_f function to generate the code for the "not found" case.

3. When the $found\_f$ or not_found_f functions are invoked, then the compiler returns the appropriate AST's that it wants placed in the "found" and "not found" positions in the instantiated code. The compiler recursively performs instantiation on these AST's before returning them.

4. When invoking the $found\_f$ function, the instantiation function passes *low* as the expression AST for the result of the access method.

Although slightly unconventional, this high-order approach to interfacing the compiler and the black-box instantiation functions is much simpler and cleaner than the alternatives.

In order to implement this scheme, the instantiator must contain functionality similar to the following:

$code' :=$ function $Instantiate(code, \mathcal{F})$

$\quad \ldots$
$\quad$ -- *Handle "if* $A_{\texttt{name}}(args)$ *then* $S1$ *else* $S2$*".*
$\quad code' := A.instOf(\texttt{name})(args,$
$\qquad\qquad \lambda e.(Instantiate(S1, \mathcal{F}[A_{\texttt{name}}[args] \mapsto e])),$
$\qquad\qquad \lambda().(Instantiate(S2, \mathcal{F})));$

end function

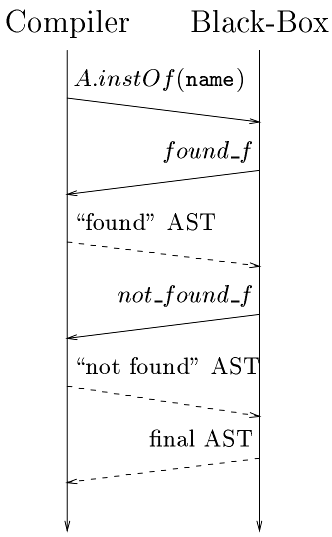


Figure 13.1: The higher order protocol in action

In this code, *code* is the original low-level plan and $\mathcal{F}$ is a set of substitutions that are to be performed during instantiation.

The sequence of function calls and returns that occur during this instantiation are illustrated in Figure 13.1. The solid arrows denote function invocation, and each is labeled with the name of the function being invoked. The dashed arrows denote function return.

13.3.3 Other considerations

In the discussion above, we considered access methods that had the following form,

if $A_{\texttt{name}}(\ldots)$ then
    S1: $\ldots$ $A_{\texttt{name}}[\ldots]$ $\ldots$
else
    S2: $\ldots$
end if

However, there are some situations when references can occur outside of these conditionals. This can happen, for instance, when the reference to A is strong.

```
S1: ... A_name[...] ...
```

If the access to A were to fail, then, in fact, the reference was *not* strong. If this situation occurs, then the reference was incorrectly marked as being strong, and there is no sense in continuing the execution of the program. So, to handle this situation, we instantiate the code as if it had been written as,

```
if A_name(...) then
    S1: ... A_name[...] ...
else
    fail;
end if
```

13.4 Generating object code

At this point in the process, we have transformed the original dense program into a set of queries, scheduled those queries into high and then low-level plans, and finally instantiated abstract access methods with their corresponding implementation. We have done this entirely within, BMF, the input and intermediate language used by the Bernoulli sparse compiler (BMF is described in Appendix B.). It remains, however, to produce object code from the BMF code of the final sparse implementation.

One approach would be to implement a compiler that transformed BMF directly into native machine code. This would not be profitable, because in order to obtain performance comparable with existing commercial compilers, we would have to invest an enormous amount of time developing such a compiler. And, such performance is vital in order to demonstrate that our sparse compiler produced code that ran as well as hand-written code.

A better approach is to use a commercial compiler directly. That is, we write a small translator from BMF to some conventional language, and then use a commercial compiler to compile this conventional language into native machine code. This is a much more economical approach to generating object code from BMF. But, what conventional language should we target?

There are two "families" of languages for which good compilers are available for the high performance architectures that we are targeting. These are the C/C++ family ([76],[116]) and the FORTRAN family ([5]). For the purposes of being a compiler "target" language, each has its advantages and disadvantages.

C provides a very rich set of programming constructs. It directly provides structures and pointers, and the C library provides explicit and portable memory management through `malloc` and `free`. C is also easy to generate, because, apart from preprocessor directives, it is a free-form language.

FORTRAN 77, while not as semantically rich as C, generally yields better performance for similarly written numerical code. This is primarily for two reasons. The first is that the FORTRAN language specifies very strict rules specifying what sort of aliasing is allowed in the source program, while C allows practically arbitrary aliasing. This aliasing often prevents aggressive reordering of loads and stored to memory. The second reason is that generating high performance object code from numerical codes has been a priority for FORTRAN compilers from day one, while this is relatively recently priority for C compilers.

In order to demonstrate the differences in performance between compiler generated C and Fortran, we wrote MVM for various sparse matrix storage formats in both C and FORTRAN. Since the aim was to compare the performance of compiler generated code, we tried to keep the codes as simple as possible. In particular, there are many optimizations that are commonly applied by a human C programmer that cannot be trivially applied and which are not immediately expressible in FORTRAN. We did not perform these optimizations in the C code. The sparse MVM code for Coordinate storage shown in Figure 13.2 illustrates the sort of code that was written.

The performance of sparse MVM for various format in C and FORTRAN is shown in Table 13.1. These runs were made on a single wide node of an SP-2, and the performance is shown in mflops. The native `xlC` and `xlf` compilers were with similar optimization flags were used. Notice that the FORTRAN performance is almost always greater than the C performance, and usually by 25% or more.

In the current implementation of the Bernoulli compiler, we chose to generate C instead of FORTRAN. We did this because, first, we were able to implement a C back-end much more quickly than a FORTRAN back-end, and, second, we were able to obtain the performance levels that we required from the generated code. While the current back-end generates C, we have

```
for (i=0; i<m; i++)
    y[i] = 0.0;
for (k=0; k<nzs; k++)
    y[rowindex[k]] = y[rowindex[k]] + value[k] * x[colindex[k]];
```

(a) In C

```
      do 10 i = 1, m
         y(i) = 0.0
 10   continue
      do 20 k = 1, totalnzs
         y(rowindex(k)) = y(rowindex(k)) +
     $       value(k) * x(colindex(k))
 20   continue
```

(b) In FORTRAN

Figure 13.2: Similar implementations of Coordinate storage MVM

Table 13.1: Sparse MVM in C and Fortran, in mflops

	C		Fortran	
Layout	$y = A * x$	$y = A^T * x$	$y = A * x$	$y = A^T * x$
coor	10.20	23.40	15.37	30.62
diag	44.80	44.55	57.23	58.51
itpack	20.26	22.46	35.64	28.93
crs	26.31	23.33	24.81	25.13
jdiag	24.65	22.86	6.55	37.87

(a) 10×10 mesh

	C		Fortran	
Format	$y = A * x$	$y = A^T * x$	$y = A * x$	$y = A^T * x$
coor	14.77	22.53	12.93	28.40
diag	54.16	55.03	86.97	87.67
itpack	19.91	20.34	37.14	25.98
crs	32.74	28.27	41.27	35.89
jdiag	27.62	24.34	7.54	46.37

(b) $10 \times 10 \times 10$ mesh

	C		Fortran	
Format	$y = A * x$	$y = A^T * x$	$y = A * x$	$y = A^T * x$
coor	14.46	22.94	14.51	29.31
diag	41.66	41.59	59.26	59.62
itpack	20.30	17.48	31.93	22.03
crs	31.91	27.91	42.07	38.63
jdiag	25.45	23.29	4.65	39.73

(c) $10 \times 10 \times 10 \times 10$ mesh designed it to generate object code that can easily to be linked with other modules written in either C or FORTRAN.

In the future, we might implement a "hybrid" back-end in which portions of the object code are generated in C and other portions in FORTRAN. The kernal loop nests, for instance might be generated and compiled as small FORTRAN subroutines for maximum performance, while the high-level control structure of the program, and any portions that require constructs more easily expressed in C (e.g., memory management) would be generated as C. The final implementation would be obtained by linking the FORTRAN and C objects.

Alternatively, many of the useful language constructs found in C can now be found in FORTRAN-90 ([4]), so perhaps that might be an appropriate target language. This would depend upon the quality of the object code that the FORTRAN-90 compilers were able to generate. We plan to investigate this as well.

13.5 Running Example

Example 13.1 Dot-product

The sparse implementation obtained by instantiating the low-level plan for sparse vector dot-product will be,

$sum := 0;$
for $ii_X := 1$ to $\#X$ do
    $(found, ii_Y) := BinarySearch(indx_Y, 1, \#Y, indx_X[ii_X]);$
    if $found$ then
        $sum := sum + value_X[ii_X] * value_Y[ii_Y];$
    end if
end do

Example 13.2 MVM

The sparse implementation obtained by instantiating the low-level plan for sparse matrix-sparse vector multiplication will be,

-- *Scatter* X *to* T
for $ii_X := 1$ to $\#X$ do
    $T[indx_X[ii_X]] := value_X[ii_X];$

```
end do
for i := 1 to n do
    for jj_A := rowptr_A[i] to rowptr_A[i-1] + 1 do
        y[i] := y[i] + value_A[jj_A] * T[colind_A[jj_A]];
    end do
end do
```

13.6 Related work

As we mentioned in Chapter 7, some relational DBMS's provide mechanisms for the user to add new storage formats to the system. In order to do this, the user must provide the DBMS with implementations of the access methods that can be invoked during query evaluation. In our case, the user provides a means of *generating* these implementations during compile-time.

As we have stated, the reason for this approach is to allow the opportunity for further optimizations of the sparse implementation. This does not seem to been considered in the database literature, perhaps because the different factors that influence performance do not make it worthwhile to do so.

Our technique of composing compilation modules together into a single code generation system by passing closures via the black-box protocal is similar in spirit to the techniques described in [55]. In that work, closures are used to encapsulate state at run-time instead of compile-time, but the ideas of using closures for composition are very similar.

13.7 Summary

In this chapter, we discussed instantiation, the last step in the sparse compilation process. We illustrated how the instantiation functions provided by the black-box protocal are used to replace stream and singleton access methods with their respective implementations. The approach used for instantiating singleton access methods is particularly novel because of its higher-order nature.

In addition, we discussed how object code might be generated for the final sparse implementation. A commercial sparse compiler would almost certainly have a native code back-end, but building such a back-end is not a good use of our research time. Therefore, we built a back-end that translates BMF into a conventional language, so that a commercial compiler can be used to generate the final object code. We discussed the merits and faults of using both C and FORTRAN for this purpose.

Part III

Extensions

Chapter 14

Dense Compilation

In this chapter we will discuss how to handle dense computations that arise in otherwise sparse codes. We start out by illustrating how these dense computations arise and point how why they are important for performance. Next, we outline various ways that a sparse compiler can generate efficient code for these dense computations. We will argue that a hybrid sparse/dense compiler seems to be the most reasonable approach to this problem. We will then discuss how the sparse compiler design presented previously can be adapted to being a hybrid compiler design. Finally, we will discuss the optimizations that we have implemented for dense codes and discuss other designed that might have been used.

14.1 Dense computations in sparse codes

14.1.1 Why dense computations arise in sparse codes

In practice, sparse programs often contain portions of code and data that are dense. In some cases, the kernels of these codes can be implemented using level 2 or 3 BLAS routines. Suppose, for instance, that a linear triangulation is used to discretize the space of a problem, and a set of four or more coupled equations are solved at each grid point (e.g., pressure, temperature, and components of a velocity vector). In this situation, four variable, or components, will be associated with each grid point, and their values will be constrained by each other and the four components at the other two vertices of the triangle. If we think of each component as a node of a graph and each constraint as being an edge, then these will be twelve nodes in this graph, and there will be edges between all pairs of these nodes. This subgraph forms a clique, as can be seen in Figure 14.1.

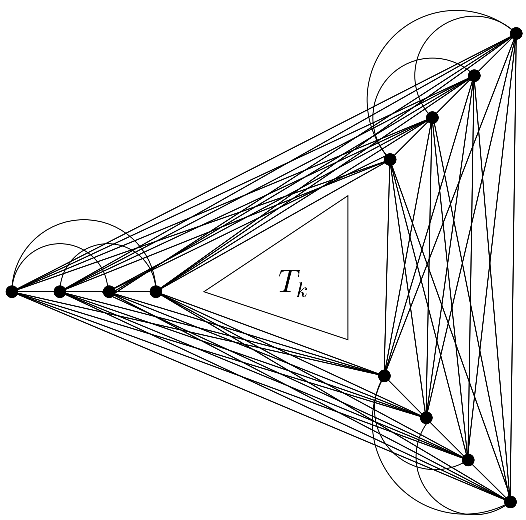


Figure 14.1: A clique in a multiple component graph

Each edge from this clique will correspond to a non-zero in the linear system, A, and if the twelve components are assigned consecutive rows and columns of A, then all of these non-zeros will form a dense 12×12 matrix along the diagonal of A. The same will be true for every other clique that is stored in consecutive rows and columns of A.

In order to obtain maximum performance when performing operations, such as MVM, on A, it has been found that it is necessary to

- Store the dense blocks of A in contiguous storage, instead of non-contiguously, as would be done in CRS or CCS storage.
- Use dense codes (i.e., simple `for` loops and simple array accesses) on these blocks.

The reason why this is necessary for top performance is that today's superscalar RISC architectures perform best under these conditions. The BlockSolve data structures ([72, 73]) were designed around these two criteria, and for certain problems there is a 50% improvement in performance when using the BlockSolve data structures and codes instead of equivalent non-blocked versions. This can be seen by examining the results in Tables 1.1 and 1.2.

14.1.2 Approaches to handling dense computations

One of the core assumptions of our work is that the user selects the storage formats for the sparse matrices. Thus, we do not see it as the compiler's responsibility to reformat the user data in order to expose dense blocks. For instance, how is the compiler to know at compile-time whether or not there are large dense blocks that are worth exposing? If there are such dense regions within a sparse matrix, then it is the user's responsibility to select an appropriate data structure that stores these regions in a dense manner.

So the real question is this: given the user's storage format selections and the original computation, how does the compiler generate the best possible dense computation?

Using a conventional optimizing compiler. The simplest approach that we might take is to generate the sparse code, as previously described in this thesis, and then to use a conventional optimizing compiler to produce the final object code. Many of the optimizations implemented in such a compiler are, in fact, targeted to dense codes, so this approach should yield high quality dense computations.

However, these compilers sometimes do not produce as high quality code as can be obtained by hand. Consider, for instance, the code obtained for MVM using the BlockSolve data structure using this approach vs. hand-written code. The performance of both in mflops is shown in Table 14.1. It is clear that the hand-written code performs significantly better the naively generated code. We conclude that, although an optimizing compiler can generate good quality code from the sparse implementation, we will need to implement certain optimizations ourselves in order to obtain performance equivalent to hand-written codes.

Implementing dense optimizations within the sparse framework. As we discussed in Section 10.7, there are many similarities between the permuted echelon form and other linear algebra frameworks developed for optimizing dense code. Perhaps, we could implement dense optimizations

Table 14.1: Hand-written vs. Naively generated BlockSolve MVM, in mflops

Grids			Mflops	
d	n	c	Hand-written	Naively generated
2	10	1	2.491	3.828
2	10	3	17.540	12.971
2	10	5	23.586	15.433
2	10	7	25.882	15.620
2	17	1	3.294	3.916
2	17	3	14.660	11.529
2	17	5	21.027	14.243
2	17	7	25.379	15.443
2	25	1	3.275	3.996
2	25	3	14.275	11.151
2	25	5	20.996	14.039
2	25	7	25.533	15.999
3	10	1	4.160	4.526
3	10	3	16.591	12.402
3	10	5	23.150	15.658
3	10	7	27.307	17.573
3	17	1	4.205	4.193
3	17	3	16.249	12.311
3	17	5	22.885	15.369
3	17	7	27.398	17.636
3	25	1	4.158	4.100
3	25	3	16.435	12.313
3	25	5	23.668	15.741
Name			Hand-written	Naively generated
`662_bus`			2.622	3.024
`685_bus`			3.356	3.736
`1138_bus`			2.511	2.770
`arco`			16.772	18.989

directly in the sparse work. Access normalization ([92]), for instance, seems to be readily implemented using our linear algebra framework. However, there are other important optimizations that do not fit so nicely in our framework. Loop tiling ([121]), which is not a linear loop transformation, is one such example.

A hybrid approach. Instead of trying to reformulate all necessary dense optimizations within the sparse framework, we could build a hybrid compiler. That is, we could build a compiler that generates the sparse portion of the code using the sparse framework and the dense portion of the code using a conventional dense framework. The advantage of this approach over implementing dense optimizations in the sparse framework is that we do not have to invest precious research time reinventing the dense wheel in the sparse framework.

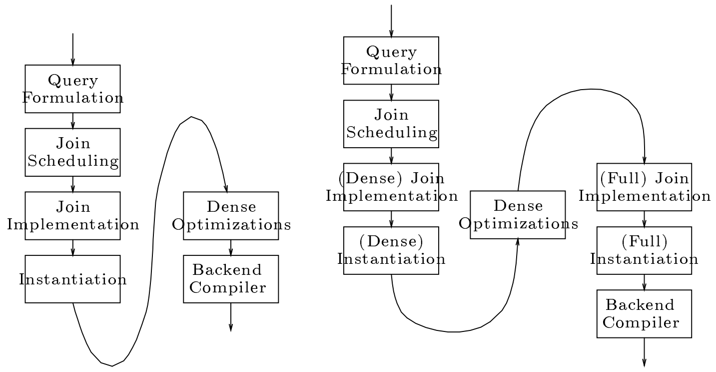
14.2 Structuring a hybrid compiler

The most straightforward approach to structuring a hybrid compiler is to simply "concatenate" the dense optimizations onto the end of the existing sparse compiler, as shown in Figure 14.2(a). While simple to implement, this approach has a fundamental defect. The code that is presented to the dense optimizations is the result of join scheduling, join implementation, and instantiation. These passes discard a tremendous amount of semantic information that is potentially useful when performing dense optimizations.

We can illustrate this point by considering an example. The following fragment of code might appear in a low-level plan after performing join implementation and before instantiation:

if $v_{\texttt{search\_i}}(i')$ then
  ...
end if

If we make some simple assumptions about the termination and side-effects of such an access method, then conventional dependence analysis ([11], [121]) can produce dependence information that is not overly conservative. However, if the access method is implemented using a binary search, then the code after instantiation might be,



(a) Concatenating (b) Interleaving

Figure 14.2: Two approaches to building a hybrid compiler

```
i := i';
low := 1;
hig := #i's;
while (low < hig) do
    mid := (low + hig)/2;
    if i ≤ vind[mid] then
        hig := mid;
    else
        low := mid + 1;
    end if
end do
if i = vind[low] then
    ...
end if
```

This code is much more difficult for conventional dependence analysis techniques. For a starter, there is no way of determining that the while loop terminates. While it might be reasonable to assume that an access method terminates, it is generally not reasonable to assume that arbitrary loops terminate.

To summarize, by not instantiating the access methods, or even by not performing join implementation, any sort of complexity that might reduce the accuracy of analysis tools can remain hidden. However, if we do not perform join implementation or instantiation, how do we expose the dense portions of the computation for dense optimization?

The solution is to *selectively* perform join implementation and instantiation. In such a scheme, join implementation and instantiation are performed twice. The first run, or *dense run*, only those portions of the code that gives rise to dense computation are processed. The portions of the code that are not dense are left unchanged. The result is code that is a mixture of dense loops and array accesses, as well as, unscheduled joins and access methods. Dense optimizations can then be performed on this hybrid code. Afterwards, join implementation and the instantiator can be run over the code again, for what is called the *sparse run*, to produce the final code for the back-end compiler. A schematic of this approach, which is the one used in the current compiler, is shown in Figure 14.2(b).

There are two issues that remain to be addressed before discussing the optimizations.

- How can join implementation and instantiation be selectively performed?
- What are the assumptions that need to be made in order to simplify dependence analysis?

Once these have been addressed, we can discuss the dense optimizations themselves.

14.3 Selectively exposing dense codes

The key to selectively performing join implementation and instantiation in order to expose dense computation is being able to recognize the joins and access methods in the high-level and low-level plans. Once these have been recognized, selective join implementation and instantiation are possible in order to expose this dense code. The approach that we take to recognizing these is to add additional information to the black-box protocol. We will describe these extensions and describe how they can be used to selectively perform join implementation and instantiation.

14.3.1 Additions to the black-box protocol

There are two additions that have been made to the black-box protocol that facilitate the recognition of dense codes. The first is, `am_lookup_type`, an alternative to the singleton instantiation function type described in Section 13.3. `am_lookup_type` is described in detail in Section C.1.4, but we can summarize this "dense" type signature as,

$A.instOf$(`name`):
  function taking
    $args$: list of $\mathcal{E}$,
    $found\_f$: function taking $\mathcal{E}$
      and returning $\mathcal{S}$,
  and returning
    $code$: $\mathcal{S}$

If this signature is compared with the previous signature given for `am_search_type`, the reader will notice that the not_found_f argument is missing. This signature is used to represent accesses that are always found. Dense array references are an instance of such an access. This might seem odd; certainly a dense array access is "not found" if the index is out of the dense array bounds. However, since the access method is only defined over the valid range of array indices, this is not an issue. For this reason, the access is always "found".

The second addition, `am_range_type`, is an alternative to the stream instantiation function type described in Section 13.2. `am_range_type` is described in detail in in Section C.1.5. Instead of providing a implementations of each of the steam methods (i.e, $h.init()$, $h.valid()$, etc.) for obtaining such a stream of values, this alternative signature provides the lower and upper bounds of the values. These can be used to construct a simple for loop for enumerating them. Here is the alternative signature,

$A.instOf$(`name`):
  function taking
    $args$: list of $\mathcal{E}$,
    $range\_f$: function taking
      lb_ex: $\mathcal{E}$,
      ub_ex: $\mathcal{E}$
      and returning $\mathcal{S}$

and returning
code: $\mathcal{S}$

This signature is higher order, as are the signatures for singleton instantiation functions. *range_f* is a function, provided by the compiler, that the black-box will call to provide the lower bound and upper bound expressions, *lb_ex* and *ub_ex* respectively to the compiler. Note, the instantiation function does not generate the loop; it simply generates the bounds. It is the responsibility of the instantiator to generate the loop from these bounds.

14.3.2 Dense join implementation

In order to understand how the joins are selectively processed during the "dense" run of the join implementer, we must first partition the join terms of a join into three sets.

Lookup terms. Those join terms that are lookup terms. These terms will be implemented using singleton access methods.

Trivial terms. Those join terms that are simple enum terms whose associate stream access method has the `am_range_type` signature described above. These are the terms will be implemented using simple loops.

Non-trivial terms. All other kinds of join terms.

The rule applied in order to selectively process a join is this: lookup terms and trivial terms are processed, and non-trivial terms are not. So, if a join consists of only lookup or trivial terms, then the "dense" run of the join implementer will process the join and will produce an appropriate low-level plan. If a join contains any non-trivial terms, then only the lookup or trivial terms of that join are processed. This policy has the effect of exposing any dense loops. It also exposes any join terms that are trivially handled by the join implementer.

14.3.3 Dense instantiation

The "dense" run of instantiation attempts to expose the dense primitive array references and simple for loops. This is done by performing the following instantiations during the "dense" run.

- If the following fragment is encountered,

  if $A_{\texttt{name}}(\ldots)$ then
      ...
  end if

  and the instantiation function $A.instOf(\texttt{name})$ has the `am_lookup_type` signature discussed above, then the test is unnecessary, since the access will always be found and the test is always true. So $A_{\texttt{name}}(\ldots)$ is replaced with $\#t$, and any dead code is eliminated.

- If an access, $A_{\texttt{name}}[\ldots]$, appears in the code, and
  - the instantiation function $A.instOf(\texttt{name})$ has the `am_lookup_type` signature discussed above, and
  - the black box specified that this access method has $O(1)$ cost,

  then the reference is instantiated during the "dense" run of the instantiator.

- If either of the following fragment is encountered,

  for $v \in A_{\texttt{name}}(\ldots)$ do
      ...
  end do

  if $v \in A_{\texttt{name}}(\ldots)$ then
      ...
  end if

  and the instantiation function $A.instOf(\texttt{name})$ has the `am_range_type` signature discussed above, then these fragments will be transformed into something like the following,

  for $v := lb$ to ub do
      ...
  end do

```
if lb ≤ v ≤ ub then
    ...
end if
```

14.3.4 A different approach

The problem with our approach is that the additions to the black-box protocol are indirect descriptions of dense code features. That is, a primitive array reference is described as a $O(1)$ cost access method whose instantiation function has the `am_fun_lookup` type signature, but a $O(1)$, `am_fun_lookup` instantiation function does not guarantee a primitive array reference. The underlying data structure could be a hash table, for instance. Similarly, a stream instantiation function with a `am_range_type` signature guarantees a simple `for` loop, but it does not guarantee that the loop will have simple bounds. In other words, our approach may be over eager, and the "dense" run of instantiation may expose more than it should.

A different approach, and one that will not expose more than it should, is to tentatively implement the joins and perform the instantiation. That is, the compiler implements joins and performs instantiation. If the results are dense code, then the dense code is used. If the results are not dense code, then the results of join implementation and instantiation are discarded, and the code is left in its original form. The advantage of this approach is that it does not require any additions to the black box protocol. The disadvantage of this approach is that it is more difficult to implement. In particular, it requires

- Mechanisms for keeping track of the original code while tentatively performing join implementation and instantiation, and
- A heuristic for recognizing dense code.

14.4 Aliasing and side-effects

In order to perform dependence analysis on the hybrid program we need to know,

- what other variables might be affected when a variable is assigned, and

- what the side-effects are of the operations that have been introduced by the compiler, such as joins and access methods.

We will address each of these in turn.

14.4.1 Aliasing

Two variables with different are said to be *aliased*, if they refer to the same memory location. In this case, a write to one variable will appear to change the value of the other. In general, computing exact aliasing between variables is undecidable, so we are interested in computing when two variables might be aliased.

Aliasing can arise in programming languages that have, for instance,

- Reference parameters, and
- Pointer variables and operations for computing the addresses of variables.

BMF has both of these.

For reference parameters, we make the same sort of aliasing assumptions that are made in FORTRAN ([5]). That is, we assume that reference parameters that are read-only within a function may be aliased to any other parameter or global variable that is read-only within that function. If a parameter or global variable is written to, then we assume that it has no aliases. Basically, these assumptions allow us to not consider aliasing that arises because of parameter passing. We should note that these are very restrictive assumptions, and are not the assumptions made by other language (e.g., C ([76])).

In BMF, we do not have pointer variables or an operator for computing addresses of variables, but we allow variables to be used in a sparse matrix's storage format declaration, which has a similar effect. However, the user provides a storage format annotation on each sparse matrix declaration that gives the names of its the storage format and the variables that comprise the underlying storage. This has the same effect as storing a pointer to one variable within another.

Consider, for instance the following sparse matrix declaration expressed in BMF,

```
var A : array int^2 of real
        << << "sparse" >>
            << storage: "crs" n: (n) nzs: "anzs" rowptr: "arowtpr"
                colind: "acolind" value: "avalue" >> >>;
```

In this case, the sparse matrix, A, is to be implemented using the "crs" black-box. The quotation syntax used for the annotation arguments, "*anzs*", "*arowtpr*", "*acolind*", and "*avalue*", indicates that these arguments are the names of variables comprise the storage of A. The parenthesis syntax used for (n) indicates that only the value and not the storage of n is needed.

In BMF, there is nothing to prevent one variable from being used as part of the underlying storage of two sparse matrices. This might happen if the two sparse matrices have the same non-zero structure, but different values.

Suppose that the sparse matrix, B, has the same non-zero structure as A and is also stored in CRS. Then it might also use the *anzs*, *arowptr*, and *acolind* variables to store its index structure and a new vector, *bvalue*, to store its values. However, if A is assigned to, then what other variables might have their values changed? Or, if one of the underlying storage variables is written, then what sparse matrices might have their values changed? Because BMF allows a sparse matrix to be specified as part of the underlying storage of another sparse matrix, the aliasing relationship can be extremely convoluted.

We have no hope of computing an accurate picture of what variables are affected by writes, because we have no idea what purpose each variable within each sparse matrix format. The great advantage of the black-box protocol is that it hides most of the details of a sparse storage format. The disadvantage, in this instance, is that it hides too much.

We have to specify a set of rules that we will use for computing what variables might be affected by every write. We have aimed to make these rules reasonable and non-intrusive. Still, when the programmer is implementing black-boxes, they should bear these rules in mind. Instead of phrasing this rule in terms of what variables are aliased, it is phrased in terms of what other variables are *invalidated* when a variable is written. The reason for this is that this "invalidates" relation is not symmetric. That is, a write to one variable may invalidate another, but not vice versa.

When a write is performed to a variable,

- The variable is invalidated.
- Each variable that comprises the storage of the written variable is invalidated, and each variable that comprises the storage of these storage variable are invalidated, and so on.
- All sparse variables in which the written variable is part of the underlying storage are invalidated, and all the variables in which these sparse variables appear as storage are invalidated, and so on.

Example 14.1 Here are some examples,

var x : ... << << storage: "bb_name" arg1: "x'" arg2: "y'" ... >> >>
var A : ... << << storage: "bb_name" arg1: "x" arg2: "y" ... >> >>;
var M : ... << << storage: "bb_name"' arg: "A" arg: "B" ... >> >>;

Writing to A invalidates x', y', x, y, A, and M, but not B. Writing to x invalidates x', y', x, A, and M, but not y and B.

14.4.2 Side-effects and access methods

A singleton access method returns a reference, not a value. So, when talking about the side-effects caused by access methods, we need to consider two situations,

- The execution of the access method to get the reference.

  At the moment, we assume that the execution of *any* access method will not produce any side-effect. This precluded data structures, like heaps, that employ path compression during searches. It would be a very minor modification to the black-box protocol to differentiate between "side-effect free" access methods, and "causes side-effects" access methods. In the latter case, we would use the invalidation rules described above to conservatively estimate what is invalidated by these methods.

- An assignment to this reference.

  When this reference is assigned to, then the invalidation rules described above are used to estimate what other variables might be affected. If the reference is somehow obscured (e.g., it is passed as an argument to a procedure), then we can use conventional dependence techniques for conservatively estimating the side-effects (e.g., interprocedural analysis ([30], [39], [121])).

A stream access method returns a value, so in this case, we only have to consider the side-effects generated by the execution of the access method.

14.5 The current dense optimizations

Since we are primarily interested in comparing the code generated by our compiler with hand-written code, and since the BlockSolve library is, by far, the most complex hand-written sparse code that we found, the dense optimizations that we have implemented are targeted to achieving the same performance as the BlockSolve codes.

Examining these codes, we found that much of the performance gained by this code over more naively written code is attributable to two basic optimizations. The first optimization was that essentially all important computational loops were replaced by calls to equivalent BLAS routines. The second optimization was that super efficient, hand-written and hand-optimized versions of the most important BLAS routines were provided by the library writers.

In order to achieve the same level of performance as this hand-written code,

- We use pattern matching to replace as many of the important dense loop nest as possible with equivalent BLAS routine calls, and

- We use the same super efficient BLAS routines implemented for the BlockSolve library.

Using this approach, we were able to obtain performance that was virtually identical to that of the handwritten code. These results can be found in Table 17.3.

This may seem like a dishonest way of achieving performance comparable to hand-written code, but we do not believe that it is. First, the primary focus of this thesis is forming and optimizing queries that describe sparse computation and not dense optimizations. We have been able to demonstrate that, keeping that later basically equal, our compiler can generate code that is as efficient as what the BlockSolve writers were able to do by hand. Second, preprocessing tools exist that substitute BLAS calls for FORTRAN loop nests. The purposes of doing so are to guarantee very high-performance for frequently occuring kernels, without spending an inordinate amount of time doing compiler optimizations. An instance of such a tool is that VAST-2 preprocessor ([97]).

14.6 Related work

The importance of extracting and exploiting dense blocks from sparse matrices has long been recognized as important to increasing performance ([1], [72]).

Automatic blocking of sparse computations is discussed in [78], in which methods are presented for a blocking a particular version of CRS MVM. In this work, a sparse matrix is examined for frequently occuring sparsity patterns, such as dense subblocks. Then, specially optimized versions of MVM are generated for performing the computation on these regions. This approach requires either the sparsity to be known at compile-time or the user to provide the sparsity patterns to the compiler. Neither appears to be more convenient than simply having the user directing the compiler to use the "blocksolve" storage format.

14.7 Summary

In this chapter, we have demonstrated how dense computations appear within sparse code and motivated why they must be recognized and optimized in order to obtain the highest performance from these code. We have presented several different approached for compiling dense and sparse codes together and argued for using the hybrid approach. We have developed methods for selectively exposing the dense codes to produce a partially implemented program and have presented rules and assumptions that enable analysis to be performed on these intermediate programs. Finally, we have discussed the dense optimizations that are performed within the current compiler. With these optimizations, we have been able to produce very efficient and competitive code, as we will see in Chapter 17.

Chapter 15

General Query Optimization

The two chapters that described our approach to query optimization, namely Chapters 11 and 12, assumed that all of the references to sparse matrices within a query were strong. As a result, we were able to make two assumptions which simplified the problem. These assumptions were that,

- The query's sparsity guard was conjunctive, and
- Fill, or the creation of new non-zero entries within a sparse matrix, did not occur.

In this chapter, we will discuss how non-conjunctive queries in which fill does not occur can be scheduled. In Chapter 16, we will discuss how fill can be handled.

The material in this chapter has not been implemented at the present time. It is provided here simply to show that the material presented earlier in this thesis can be extended to handled more general situations.

15.1 The problem with general queries

15.1.1 An example that works

When disjuncts appear within the query predicate or weak references appear in the body of the query, then different operators besides the inner join, $\bowtie$ must be used to express the query. Consider the following code to sum the entries of two sparse vectors,

```
sum := 0;
for i := 1 to n do
    sum := sum + X[i] + Y[i];
```

The query used to describe this computation can be expressed using the outer join operator, $\leftrightarrow$.

$sum := 0;$
for $\langle i, i_X, v_X, i_Y, v_Y \rangle \in$
$\quad\quad (I(i, i_X, i_Y) \bowtie (X(i_X, v_X) \leftrightarrow Y(i_Y, v_Y)))$ do
$\quad sum := sum + RVAL(v_X) * RVAL(v_Y)$
end do

The $\bowtie$ with $I(i)$ essentially performs "bounds checking" against the original iteration sparse, so we will ignore it for the remainder of this example.

Given that X and Y are stored in sparse vectors, the join scheduler described in Chapter 11 can be used to produce the following high-level plan.

Stage	Linear	X	Y	
1.	$Join(\mathbf{v}'_1 = \mathbf{h}'_1 t_1 + \cdots,$	$\{\{X.i, X.ii\},$	$\{Y.ii, Y.i\}\}$	$)$
2.	$Unjoinable($	$\{X.v\}$		$)$
3.	$Unjoinable($		$\{Y.v\}$	$)$

where the join operation in Stage 1 is to be $\leftrightarrow$.

We did not describe outer join implementations in Chapter 12, but such implementations exist. One such implementation uses three loops to compute each of $BVAL(X.v) \wedge BVAL(Y.v)$, $BVAL(X.v) \wedge \neg BVAL(Y.v)$, and $\neg BVAL(X.v) \wedge BVAL(Y.v)$, which together form $BVAL(X.v) \wedge BVAL(Y.v)$, the original sparsity guard of the query.

$sum := 0;$
-- *Compute* $BVAL(X.v) \wedge BVAL(Y.v)$
for $ii_X \in X_{\texttt{enum\_ii}}[\,]$ do
$\quad i := X_{\texttt{lookup\_i}}[ii_X];$
$\quad$ if $Y_{\texttt{search\_i}}(i)$ then
$\quad\quad ii_Y := Y_{\texttt{search\_i}}[i];$
$\quad\quad sum := sum + X_{\texttt{lookup\_v}}[ii_X] + Y_{\texttt{lookup\_i}}[ii_Y];$

```
    end if
end do
-- Compute BVAL(X.v) ∧ ¬BVAL(Y.v)
for ii_X ∈ X_enum_ii[ ] do
    i := X_lookup_i[ii_X];
    if ¬Y_search_i(i) then
        sum := sum + X_lookup_v[ii_X];
    end if
end do
-- Compute ¬BVAL(X.v) ∧ BVAL(Y.v)
for ii_Y ∈ Y_enum_ii[ ] do
    i := Y_lookup_i[ii_Y];
    if ¬X_search_i(i) then
        sum := sum + Y_lookup_v[ii_Y];
    end if
end do
```

15.1.2 An example that does not work

However, suppose that instead of being stored in sparse vectors, X and Y were stored in a *set* of sparse vectors. That is, suppose that X and Y each have an additional field, d, that partitions the entries of each sparse vector into arbitrary disjoint subsets, which are each stored as a sparse vector. In this case, the hierarchy of indices produced for each of X and Y might be,

$$\{X.d\} \rightarrow \{X.i, X.ii\} \rightarrow \{X.v\}$$
$$\{Y.d\} \rightarrow \{Y.i, Y.ii\} \rightarrow \{Y.v\}$$

In this case, the join scheduler described in Chapter 11 might produce the following high-level plan.

Stage	Linear	X	Y	
1.	$Unjoinable($	$\{X.d\}$		$)$
2.	$Unjoinable($		$\{Y.d\}$	$)$
3.	$Join(\mathbf{v}'_1 = \mathbf{h}'_1 t_1 + \cdots,$	$\{\{X.i, X.ii\},$	$\{Y.ii, Y.i\}\}$	$)$
4.	$Unjoinable($	$\{X.v\}$		$)$
5.	$Unjoinable($		$\{Y.v\}$	$)$

Then, the join implementer will generate the following low-level plan from this high-level plan.

```
sum := 0;
for d_X ∈ X_enum_d[ ] do
    for d_Y ∈ Y_enum_d[ ] do
        -- Compute BVAL(X.v) ∧ BVAL(Y.v)
        for ii_X ∈ X_enum_ii[d_X] do
            i := X_lookup_i[ii_X];
            if Y_search_i(d_Y, i) then
                ii_Y := Y_search_i[d_Y, i];
                sum := sum + X_lookup_v[ii_X] + Y_lookup_i[ii_Y];
            end if
        end do
        -- Compute BVAL(X.v) ∧ ¬BVAL(Y.v)
        for ii_X ∈ X_enum_ii[d_X] do
            i := X_lookup_i[ii_X];
            if ¬Y_search_i(d_Y, i) then
                sum := sum + X_lookup_v[ii_X];
            end if
        end do
        -- Compute ¬BVAL(X.v) ∧ BVAL(Y.v)
        for ii_Y ∈ Y_enum_ii[d_Y] do
            i := Y_lookup_i[ii_Y];
            if ¬X_search_i(d_X, i) then
                sum := sum + Y_lookup_v[ii_Y];
            end if
        end do
    end do
end do
```

But this code does not compute the correct results! Suppose that X and Y are following relations,

$X.i$	$X.d$	$X.v$
1	1	a

$Y.i$	$Y.d$	$Y.v$
1	1	p
2	2	q

The natural outer join of these two relations is,

$X \leftrightarrow_i Y = R =$

$R.i$	$R.d_X$	$R.v_X$	$R.d_Y$	$R.v_Y$
1	1	a	1	p
2	ω	ω	2	q

However, consider what happens when the example code is executed.

- $d_X = 1, d_Y = 1$:
  - Compute $BVAL(X.v) \wedge BVAL(Y.v)$

    $\langle i = 1, X.v = a, Y.v = p \rangle$
  - Compute $BVAL(X.v) \wedge \neg BVAL(Y.v)$

    $\langle \rangle$
  - Compute $\neg BVAL(X.v) \wedge BVAL(Y.v)$

    $\langle \rangle$
- $d_X = 1, d_Y = 2$:
  - Compute $BVAL(X.v) \wedge BVAL(Y.v)$

    $\langle \rangle$
  - Compute $BVAL(X.v) \wedge \neg BVAL(Y.v)$

    $\langle i = 1, X.v = a, Y.v = \omega \rangle$
  - Compute $\neg BVAL(X.v) \wedge BVAL(Y.v)$

    $\langle i = 2, X.v = \omega, Y.v = q \rangle$

The error occurs when the $\langle i = 1, X.v = a, Y.v = \omega \rangle$ is generated. This tuple is not in the result of $X \leftrightarrow Y$ as shown above.

15.1.3 The nature of the problem

Recall that the outer join and left outer join operators can be described in terms of the inner join and anti-join operators,

$$R \leftrightarrow S = (R \bowtie S) \cup (R \triangleright S) \cup (S \triangleright R)$$
$$R \rightarrow S = (R \bowtie S) \cup (R \triangleright S)$$

Theorem 11.1 showed that the following equality holds,

$$B = \biguplus_k B_k \Rightarrow A \bowtie B = \biguplus_k (A \bowtie B_k)$$

however, the equivalent equality for the $\triangleright$ operator does not hold.

$$B = \biguplus_k B_k \Rightarrow A \triangleright B \neq \biguplus_k (A \triangleright B_k)$$

Unfortunately, the join scheduler described in Chapter 11 are centered around this partitioning being true. In particular the join scheduler assumes that any partitioning of a relation can safely be exploited for scheduling. However, this is only true for the $\bowtie$ operator and not the $\leftrightarrow$ operator. So, the high-level plan given in Section 15.1.2, which uses just such a partition on $X.d$ and $Y.D$, is incorrect.

15.2 Safe nesting of general queries

15.2.1 Statement of partitioning theorem

A different partitioning theorem is required for handling the $\leftrightarrow$ and $\rightarrow$ operators. This theorem will also work for the $\bowtie$ operator so it is complete enough to serve for as the basis for a general query optimization framework. In this section, we will refer to $\bowtie$, $\leftrightarrow$ or $\rightarrow$ collectively as *join operators* or simply *joins*.

The partitioning equality that we will need for performing query optimization on any of the three join operators in given by the following theorem,

Theorem 15.1 *Let A and B be two relations with the schemata $\langle i, j, a\rangle$ and $\langle i, j, b\rangle$ respectively, then*

$$A \ op \ B = \biguplus_{i' \in (\pi_i A) \ op \ (\pi_i B)} \{\langle i' \rangle\} \times ((\pi_{j,a}\sigma_{i=i'} A) \ op \ (\pi_{j,b}\sigma_{i=i'} B))$$

where op is one of $\bowtie$, $\leftrightarrow$ or $\rightarrow$.

The proof of this theorem appears at the end of this section.

Notice that only four fields are used in the relations appearing in this theorem,

- i and j are the two join fields between A and B. The partitioning is occuring on i, and the join on j occurs within each partition.
- The fields a and b are unjoinable fields from A and B respectively.

It should be clear that, even though this theorem is not expressed in terms of relations with arbitrary fields, it can still be applied to relations with any schema because the four fields, i, j, a, and b, cover all of the situations in which fields can occur.

15.2.2 Notation

- $R_{i'} = \sigma_{i=i'} R$.

15.2.3 Partitioning a single relation

The following lemma says that is is same to partition a relation, $R(i, r)$, using selection and projection. This partitioning will form the structure of the generated loop nests.

Lemma 15.2 *Given $R(i, r)$,*

$$R = \bigcup_{i' \in \pi_i R} (\{\langle i' \rangle\} \times \pi_r R_{i'})$$

Proof

- Assume $\langle i', r' \rangle \in R$.

$$\begin{aligned}
&\langle i', r' \rangle \in R \Rightarrow i' \in \pi_i R \\
&\langle i', r' \rangle \in R \Rightarrow \langle i', r' \rangle \in R_{i'} \\
&\qquad\qquad\quad \Rightarrow r' \in \pi_r R_{i'} \\
&\qquad\qquad\quad \Rightarrow \langle i', r' \rangle \in \{\langle i' \rangle\} \times \pi_r R_{i'} \\
&\qquad\qquad\quad \Rightarrow \langle i', r' \rangle \in \textstyle\bigcup_{i'' \in \pi_i R} (\{\langle i' \rangle\} \times \pi_r R_{i'})
\end{aligned}$$

- Assume $\langle i', r' \rangle \in \bigcup_{i'' \in \pi_i R} (\{\langle i'' \rangle\} \times \pi_r R_{i''})$.

$$\begin{aligned}
&\langle i', r' \rangle \in \textstyle\bigcup_{i'' \in \pi_i R} (\{\langle i'' \rangle\} \times \pi_r R_{i''}) \\
&\qquad\qquad\quad \Rightarrow \exists i'' \in \pi_i R, \langle i', r' \rangle \in (\{\langle i'' \rangle\} \times \pi_r R_{i''}) \\
&\qquad\qquad\quad \Rightarrow \exists i'' \in \pi_i R, (i' = i'' \wedge r' \in \pi_r R_{i''}) \\
&\qquad\qquad\quad \Rightarrow i' \in \pi_i R \wedge r' \in \pi_r R_{i'} \\
&\qquad\qquad\quad \Rightarrow i' \in \pi_i R \wedge (\exists i''', \langle i''', r' \rangle \in R_{i'}) \\
&\qquad\qquad\quad \Rightarrow i' \in \pi_i R \wedge (\exists i''', \langle i''', r' \rangle \in R \wedge i' = i''') \\
&\qquad\qquad\quad \Rightarrow i' \in \pi_i R \wedge \langle i', r' \rangle \in R \\
&\qquad\qquad\quad \Rightarrow \langle i', r' \rangle \in R
\end{aligned}$$

Corollary 15.3 *Given $V(i, j, a, b)$,*

$$V_{i'} = \{\langle i' \rangle\} \times \pi_{j,a,b} V_{i'}$$

Proof This follows from Lemma 15.2, because $\pi_i V_{i'} = \{\langle i' \rangle\}$.

15.2.4 Relaxed partitioning of a single relation

Lemma 15.2 shows how a relation can be partitioned using selections and projections. Lemma 15.4 provides a more “relaxed” version of the partitioning. That is, instead of using $\pi_i R$ on the $\bigcup$, S is used, where $\pi_i R \subseteq S$. This lemma shows that performing the $\bigcup$ over S does not change its results.

Lemma 15.4

$$\pi_i R \subseteq S \Rightarrow R = \bigcup_{i' \in S} (\{\langle i' \rangle\} \times \pi_r R_{i'})$$

Proof Since, $\pi_i R \subseteq S$,

$$\begin{aligned}
&\textstyle\bigcup_{i' \in S} (\{\langle i' \rangle\} \times \pi_r R_{i'}) \\
&\quad = \textstyle\bigcup_{i' \in \pi_i R} (\{\langle i' \rangle\} \times \pi_r R_{i'}) \cup \bigcup_{i' \in S - \pi_i R} (\{\langle i' \rangle\} \times \pi_r R_{i'}) \\
&\quad = R \cup \textstyle\bigcup_{i' \in S - \pi_i R} (\{\langle i' \rangle\} \times \pi_r R_{i'})
\end{aligned}$$

If it can be shown that

$$\bigcup_{i' \in S - \pi_i R} (\{\langle i' \rangle\} \times \pi_r R_{i'}) = \phi,$$

then we are done. Suppose that it is not empty. Then $\exists \langle i', r' \rangle$ such that

$$\begin{aligned}
&\langle i', r' \rangle \in \textstyle\bigcup_{i'' \in S - \pi_i R} (\{\langle i'' \rangle\} \times \pi_r R_{i''}) \\
&\quad \Rightarrow \exists i'' \in S - \pi_i R, \langle i', r' \rangle \in (\{\langle i'' \rangle\} \times \pi_r R_{i''}) \\
&\quad \Rightarrow \exists i'' \in S - \pi_i R, i' = i'' \wedge r' \in \pi_r R_{i''} \\
&\quad \Rightarrow i' \in S - \pi_i R \wedge r' \in \pi_r R_{i'} \\
&\quad \Rightarrow i' \in S \wedge i' \notin \pi_i R \wedge r' \in \pi_r R_{i'}
\end{aligned}$$

Since $i' \notin \pi_i R$, $R_{i'} = \phi$, so

$$r' \in \pi_r R_{i'} \Rightarrow r' \in \pi_r \phi \Rightarrow \exists i'', \langle i'', r' \rangle \in \phi,$$

which is a contradiction.

15.2.5 Operations on single field relations

Lemma 15.5 *Given two single field relations, $F(i)$ and $G(i)$,*

- $F \ltimes G = F \bowtie G$
- $F \bowtie G = F \bigcap G$

- $F \triangleright G = F - G$
- $F \rightarrow G = F$
- $F \leftrightarrow G = F \bigcup G$

Proof

- $$\begin{aligned} i \in F \ltimes G &\Leftrightarrow i \in \pi_i(F \bowtie G) \\ &\Leftrightarrow i \in F \bowtie G \end{aligned}$$
- $$\begin{aligned} i \in F \bowtie G &\Leftrightarrow i \in F \wedge i \in G \\ &\Leftrightarrow i \in F \bigcap G \end{aligned}$$
- $$\begin{aligned} i \in F \triangleright G &\Leftrightarrow i \in F - (F \ltimes G) \\ &\Leftrightarrow i \in F - (F \bowtie G) \\ &\Leftrightarrow i \in F \wedge i \notin (F \bowtie G) \\ &\Leftrightarrow i \in F \wedge (i \notin F \vee i \notin G) \\ &\Leftrightarrow i \in F \wedge i \notin G \\ &\Leftrightarrow i \in F - G \end{aligned}$$
- $$\begin{aligned} i \in F \rightarrow G &\Leftrightarrow i \in (F \bowtie G) \bigcup (F \triangleright G) \\ &\Leftrightarrow i \in F \bowtie G \vee i \in F \triangleright G \\ &\Leftrightarrow i \in F \bowtie G \vee i \in F - G \\ &\Leftrightarrow (i \in F \wedge i \in G) \vee (i \in F \wedge i \notin G) \\ &\Leftrightarrow i \in F \wedge (i \in G \vee i \notin G) \\ &\Leftrightarrow i \in F \end{aligned}$$
- $$\begin{aligned} i \in F \leftrightarrow G &\Leftrightarrow i \in (F \rightarrow G) \bigcup (G \rightarrow F) \\ &\Leftrightarrow i \in (F \rightarrow G) \vee i \in (G \rightarrow F) \\ &\Leftrightarrow i \in F \vee i \in G \\ &\Leftrightarrow i \in F \bigcup G \end{aligned}$$

15.2.6 Approximating single field joins

Lemma 15.6 *Given $A(i, j, a)$ and $B(i, j, b)$,*

$$\pi_i(A \ op \ B) \subseteq (\pi_i A) \ op \ (\pi_i B)$$

Proof

- $$\begin{aligned} i' \in \pi_i(A \bowtie B) &\Rightarrow \exists j', a', b', \langle i', j', a', b'\rangle \in A \bowtie B \\ &\Rightarrow \exists j', a', b', (\langle i', j', a'\rangle \in A) \wedge (\langle i', j', b'\rangle \in B) \\ &\Rightarrow i' \in \pi_i A \wedge i' \in \pi_i B \\ &\Rightarrow i' \in \pi_i A \bigcup \pi_i B \\ &\Rightarrow i' \in \pi_i A \bowtie \pi_i B \end{aligned}$$

- $$\begin{aligned} i' \in \pi_i(A \to B) &\Rightarrow \exists j', a', b',\ \langle i', j', a', b'\rangle \in A \to B \\ &\Rightarrow \exists j', a', b', \langle i', j', a', b'\rangle \in (A \bowtie B) \bigcup (A \triangleright B) \\ &\Rightarrow \exists j', a', b', (\langle i', j', a', b'\rangle \in A \bowtie B) \vee (\langle i', j', a', b'\rangle \in A \triangleright B) \\ &\Rightarrow \exists j', a', b', (\langle i', j', a'\rangle \in A \wedge \langle i', j', b'\rangle \in B) \\ &\qquad\qquad \vee (\langle i', j', a', b'\rangle \in A \triangleright B) \\ &\Rightarrow \exists j', a', b', (\langle i', j', a'\rangle \in A \wedge \langle i', j', b'\rangle \in B) \\ &\qquad\qquad \vee (\langle i', j', a', b'\rangle \in (A - (A \ltimes B)) \times \Omega) \\ &\Rightarrow \exists j', a', b', (\langle i', j', a'\rangle \in A \wedge \langle i', j', b'\rangle \in B) \\ &\qquad\qquad \vee (\langle i', j', a'\rangle \in A - (A \ltimes B)) \\ &\Rightarrow \exists j', a', b', (\langle i', j', a'\rangle \in A \wedge \langle i', j', b'\rangle \in B) \\ &\qquad\qquad \vee (\langle i', j', a'\rangle \in A \wedge \langle i', j', a'\rangle \notin A \ltimes B) \\ &\Rightarrow \exists j', a', b', \langle i', j', a'\rangle \in A \wedge (\langle i', j', b'\rangle \in B \vee \langle i', j', a'\rangle \notin A \ltimes B) \\ &\Rightarrow \exists j', a', b', \langle i', j', a'\rangle \in A \\ &\Rightarrow i' \in \pi_i A \\ &\Rightarrow i' \in \pi_i A \to \pi_i B \end{aligned}$$

- $$\begin{aligned} i' \in \pi_i(A \leftrightarrow B) &\Rightarrow i' \in \pi_i(A \to B \bigcup B \to A) \\ &\Rightarrow i' \in \pi_i(A \to B) \vee i' \in \pi_i(B \to A) \\ &\Rightarrow i' \in (\pi_i A \to \pi_i B) \vee i' \in (\pi_i B \to \pi_i A) \\ &\Rightarrow i' \in (\pi_i A \to \pi_i B) \bigcup (\pi_i B \to \pi_i A) \\ &\Rightarrow i' \in \pi_i A \leftrightarrow \pi_i B \end{aligned}$$

15.2.7 Distributing selections

Lemma 15.7 *Given the two relations, $R(i, r)$ and $S(i, s)$,*

$$\sigma_{i=i'}(R\ op\ S) = R_{i'}\ op\ S_{i'}$$

Proof

- $\sigma_{i=i'}(R \bowtie S) = R_{i'} \bowtie S_{i'}$. From [120].

- $$\begin{aligned}\sigma_{i=i'}(R \triangleright S) &= \sigma_{i=i'}((R - (R \ltimes S)) \times \Omega) \\ &= \sigma_{i=i'}(R - (R \ltimes S)) \times \Omega \\ &= (R_{i'} - \sigma_{i=i'}(R \ltimes S)) \times \Omega \\ &= (R_{i'} - (R_{i'} \ltimes S_{i'})) \times \Omega \\ &= R_{i'} \triangleright S_{i'}\end{aligned}$$

- $$\begin{aligned}\sigma_{i=i'}(R \rightarrow S) &= \sigma_{i=i'}(R \bowtie S \bigcup R \triangleright S) \\ &= \sigma_{i=i'}(R \bowtie S) \bigcup \sigma_{i=i'}(R \triangleright S) \\ &= R_{i'} \bowtie S_{i'} \bigcup R_{i'} \triangleright S_{i'} \\ &= R_{i'} \rightarrow S_{i'}\end{aligned}$$

- $$\begin{aligned}\sigma_{i=i'}(R \leftrightarrow S) &= \sigma_{i=i'}(R \bowtie S \bigcup R \triangleright S \bigcup S \triangleright R) \\ &= \sigma_{i=i'}(R \bowtie S) \bigcup \sigma_{i=i'}(R \triangleright S) \bigcup \sigma_{i=i'}(S \triangleright R) \\ &= (R_{i'} \bowtie S_{i'}) \bigcup (R_{i'} \triangleright S_{i'}) \bigcup (S_{i'} \triangleright R_{i'}) \\ &= R_{i'} \leftrightarrow S_{i'}\end{aligned}$$

15.2.8 Distributing projections with selections

Lemma 15.8 *Given $A(i,j,a)$ and $B(i,j,b)$,*

$$\pi_{j,a,b}(A_{i'} \;\; op \;\; B_{i'}) = \pi_{j,a,b}A_{i'} \;\; op \;\; \pi_{j,a,b}B_{i'}$$

Proof

- $$\begin{aligned}\pi_{j,a,b}(A_{i'} \bowtie B_{i'}) &= \pi_{j,a,b}((\{\langle i'\rangle\} \times \pi_{j,a,b}A_{i'}) \bowtie (\{\langle i'\rangle\} \times \pi_{j,a,b}B_{i'})) && (1) \\ &= \pi_{j,a,b}(\{\langle i'\rangle\} \times (\pi_{j,a}A_{i'} \bowtie \pi_{j,b}B_{i'}) && (2) \\ &= \pi_{j,a}A_{i'} \bowtie \pi_{j,b}B_{i'}\end{aligned}$$

  Notes:

  (1) Corollary 15.3.

  (2) This can be shown by expanding the definition of $\bowtie$.

- $\pi_{j,a,b}(A_{i'} \triangleright B_{i'})$
$$\begin{aligned}
&= \pi_{j,a,b}((A_{i'} - A_{i'} \ltimes B_{i'}) \times \Omega) \\
&= \pi_{j,a}(A_{i'} - A_{i'} \ltimes B_{i'}) \times \Omega \\
&= \pi_{j,a}((\{\langle i'\rangle\} \times \pi_{j,a}A_{i'}) - (\{\langle i'\rangle\} \times \pi_{j,a}A_{i'}) \ltimes (\{\langle i'\rangle\} \times \pi_{j,b}B_{i'})) \times \Omega \\
&= \pi_{j,a}((\{\langle i'\rangle\} \times \pi_{j,a}A_{i'}) - (\{\langle i'\rangle\} \times (\pi_{j,a}A_{i'} \ltimes \pi_{j,b}B_{i'}))) \times \Omega \qquad (1) \\
&= \pi_{j,a}(((\{\langle i'\rangle\} - \{\langle i'\rangle\}) \times \pi_{j,a}A_{i'}) \\
&\qquad \bigcup(\{\langle i'\rangle\} \times (\pi_{j,a}A_{i'} - (\pi_{j,a}A_{i'} \ltimes \pi_{j,b}B_{i'})))) \times \Omega \qquad (2) \\
&= \pi_{j,a}(\{\langle i'\rangle\} \times (\pi_{j,a}A_{i'} - (\pi_{j,a}A_{i'} \ltimes \pi_{j,b}B_{i'}))) \times \Omega \qquad (3) \\
&= (\pi_{j,a}A_{i'} - (\pi_{j,a}A_{i'} \ltimes \pi_{j,b}B_{i'})) \times \Omega \\
&= \pi_{j,a}A_{i'} \triangleright \pi_{j,b}B_{i'}
\end{aligned}$$

  Notes:

  (1) The can be shown by expanding the definition of $\ltimes$.

  (2) $(R \times S) - (T \times U) = ((R - T) \times S) \cup (R \times (S - U))$.

  (3) Since $\{\langle i'\rangle\} - \{\langle i'\rangle\} = \phi$.

- $$\begin{aligned}
\pi_{j,a,b}(A_{i'} \rightarrow B_{i'}) &= \pi_{j,a,b}(A_{i'} \bowtie B_{i'} \bigcup A_{i'} \triangleright B_{i'}) \\
&= \pi_{j,a,b}(A_{i'} \bowtie B_{i'}) \bigcup \pi_{j,a,b}(A_{i'} \triangleright B_{i'}) \\
&= (\pi_{j,a}A_{i'} \bowtie \pi_{j,b}B_{i'}) \bigcup(\pi_{j,a}A_{i'} \triangleright \pi_{j,b}B_{i'}) \\
&= \pi_{j,a}A_{i'} \rightarrow \pi_{j,b}B_{i'}
\end{aligned}$$

- $$\begin{aligned}
\pi_{j,a,b}(A_{i'} \leftrightarrow B_{i'}) &= \pi_{j,a,b}((A_{i'} \bowtie B_{i'}) \bigcup(A_{i'} \triangleright B_{i'}) \bigcup(B_{i'} \triangleright A_{i'})) \\
&= \pi_{j,a,b}(A_{i'} \bowtie B_{i'}) \bigcup \pi_{j,a,b}(A_{i'} \triangleright B_{i'}) \bigcup \pi_{j,a,b}(B_{i'} \triangleright A_{i'}) \\
&= (\pi_{j,a}A_{i'} \bowtie \pi_{j,b}B_{i'}) \bigcup(\pi_{j,a}A_{i'} \triangleright \pi_{j,b}B_{i'}) \bigcup(\pi_{j,b}B_{i'} \triangleright \pi_{j,a}A_{i'}) \\
&= \pi_{j,a}A_{i'} \leftrightarrow \pi_{j,b}B_{i'}
\end{aligned}$$

15.2.9 Theorem 15.1

Proof: Theorem 15.1 Let $\dot{R}_{i'} = \{\langle i'\rangle\} \times (\pi_{j,a}A_{i'})\ \mathrm{op}\ (\pi_{j,b}B_{i'})$.

- $\dot{R}_{i'} \neq \dot{R}_{i''} \Rightarrow \dot{R}_{i'} \bigcap \dot{R}_{i''} = \phi$

  $$\begin{aligned}
\dot{R}_{i'} \neq \dot{R}_{i''} &\Rightarrow i' \neq i'' \\
&\Rightarrow \{\langle i'\rangle\} \bigcap \{\langle i''\rangle\} = \phi \\
&\Rightarrow \{\langle i'\rangle\} \times ((\pi_{j,a}A_{i'})\ \mathrm{op}\ (\pi_{j,b}B_{i'})) \bigcap \\
&\qquad\quad \{\langle i''\rangle\} \times ((\pi_{j,a}A_{i''})\ \mathrm{op}\ (\pi_{j,b}B_{i''})) = \phi \\
&\Rightarrow \dot{R}_{i'} \bigcap \dot{R}_{i''} = \phi
\end{aligned}$$

- A op $B = \bigcup_{i' \in (\pi_i A) \text{ op } (\pi_i B)} \dot{R}_{i'}$.
  Since, by Lemma 15.6,

$$\pi_i(A \text{ op } B) \subseteq (\pi_i A) \text{ op } (\pi_i B),$$

  and since, by Lemma 15.4,

$$\pi_i R \subseteq S \Rightarrow R = \bigcup_{i' \in S} \dot{R}_{i'},$$

  it follows that,

$$A \text{ op } B = \bigcup_{i' \in (\pi_i A) \text{ op } (\pi_i B)} (\{\langle i' \rangle\} \times \pi_{j,a,b}\sigma_{i=i'}(A \text{ op } B))$$

  By Lemma 15.7,

$$A \text{ op } B = \bigcup_{i' \in (\pi_i A) \text{ op } (\pi_i B)} (\{\langle i' \rangle\} \times \pi_{j,a,b}(A_{i'} \text{ op } B_{i'}))$$

  By Lemma 15.8,

$$A \text{ op } B = \bigcup_{i' \in (\pi_i A) \text{ op } (\pi_i B)} (\{\langle i' \rangle\} \times (\pi_{j,a,b}A_{i'} \text{ op } \pi_{j,a,b}B_{i'}))$$

$\square$.

15.3 Overview of the new approach

The new partitioning rule given by Theorem 15.1 provides for a very different approach to join scheduling than was described in Chapter 11. Here is basically how it will work.

1. A query will be formed from the original dense specification, as described in Chapter 8. This query will have the form,

   for $v \in \pi_v \sigma_{SP(body)}(I \to A_1 \to \cdots \to A_p)$ do
       body
   end do

   where,

   $$I(\mathbf{i}, \mathbf{a}_1, \ldots, \mathbf{a}_p) = \{\langle \mathbf{i}, \mathbf{F}_1\mathbf{i} + \mathbf{f}_1, \ldots, \mathbf{F}_p\mathbf{i} + \mathbf{f}_p\rangle | \mathbf{i} \in bounds_{loop}\}$$

   and each $\mathbf{F}_k\mathbf{i} + \mathbf{f}_k$ is the array access function to A_k. No attempt is made to identify inner joins at this point.

2. The linear framework describes in Chapter 10 is used to construct a sequence of nested join surfaces in terms of the parametric variables, t_1 through t_n.

3. If the hierarchies of indices of the relations involved in the join do not provide efficient access for the π's and σ's that are required by the new partitioning theorem, then these relations will be remapped into storage formats which do.

4. The join scheduler will use the nesting suggested by the join surfaces to determine what joins occur at each stage of the high-level plan. At each stage k of the high-level plan, we will compute the expressions,

   $$f' \in \pi_f A \qquad \pi_{\neg f}\sigma_{f=f'} A,$$

   where, $A.f = [\mathbf{H}'_k]_{A.f}\hat{\mathbf{t}}_k + [\mathbf{h}'_k]_{A.f}t_k + [\bar{\mathbf{v}}'_k]_{A.f}$.

5. The join implementer will examine each join stage of the high-level plan and determine which join operator, $\bowtie$, $\leftrightarrow$, or $\rightarrow$, can be used. Then, it will select the implementations of these operators and produce the low-level plan.

15.4 Remapping

The new partitioning rule requires that several expressions be computed that were not required by the inner join partitioning rule. For each sparse matrix, A, these expressions are,

$$f' \in \pi_f A \qquad \pi_{\neg f}\sigma_{f=f'}A,$$

where $A.f$ appears in the kth join surface. In this section, we will explore the impact of these requirements on the way in which relations are stored.

15.4.1 An example

Consider the following sparse computations

```
for i := 1 to n do
    for j := 1 to n do
        sum := sum + A[i, j] + B[i, j]
    end do end do
```

where A is in CRS and B is in CCS. Suppose that the join surfaces developed for this computation are,

$$\begin{pmatrix}1\\1\\1\end{pmatrix} t_1 = \begin{pmatrix}I.i\\A.i\\B.i\end{pmatrix} \qquad \rightarrow \begin{pmatrix}1\\1\\1\end{pmatrix} t_2 = \begin{pmatrix}I.j\\A.j\\B.j\end{pmatrix}$$

and the hierarchy of indices were,

$$\begin{aligned} A :&\{A.i\} \rightarrow \{A.jj, A.j\} \rightarrow \{A.v\} \\ B :&\{B.j\} \rightarrow \{B.ii, B.i\} \rightarrow \{B.v\} \end{aligned}$$

If we choose to schedule an $t_1 \to t_2$ nesting using the new partitioning rule, which is equivalent to the nesting $i \to j$, then we would obtain the following partitions,

$$
\begin{aligned}
A \leftrightarrow B &= \biguplus_{i' \in (\pi_i A) \leftrightarrow (\pi_i B)} \langle i' \rangle \times ((\pi_{j,a}\sigma_{i=i'}A) \leftrightarrow (\pi_{j,b}\sigma_{i=i'}B)) \\
&= \biguplus_{i' \in (\pi_i A) \leftrightarrow (\pi_i B)} \langle i' \rangle \times (\\
&\qquad \biguplus_{j' \in (\pi_j \pi_{j,a}\sigma_{i=i'}A) \leftrightarrow (\pi_j \pi_{j,b}\sigma_{i=i'}B)} \langle j' \rangle \times (\\
&\qquad\quad ((\pi_a \sigma_{j=j'}\pi_{j,a}\sigma_{i=i'}A) \leftrightarrow (\pi_b \sigma_{j=j'}\pi_{j,b}\sigma_{i=i'}B))))
\end{aligned}
$$

which can be expressed more clearly in pseudocode as,

let $A' = \{\langle i', \tilde{A}' \rangle | i' \in \pi_i A \wedge \tilde{A}' = \pi_{j,a}\sigma_{i=i'}A\}$;
let $B' = \{\langle i', \tilde{B}' \rangle | i' \in \pi_i B \wedge \tilde{B}' = \pi_{j,b}\sigma_{i=i'}B\}$;
for $\langle i', \tilde{A}', \tilde{B}' \rangle \in A' \leftrightarrow B'$ do
    let $A'' = \{\langle j', \tilde{A}'' \rangle | j' \in \pi_i \tilde{A}' \wedge \tilde{A}'' = \pi_a \sigma_{j=j'}\tilde{A}'\}$;
    let $B'' = \{\langle j', \tilde{B}'' \rangle | j' \in \pi_i \tilde{B}' \wedge \tilde{B}'' = \pi_b \sigma_{j=j'}\tilde{B}'\}$;
    for $\langle j', \tilde{A}'', \tilde{B}'' \rangle \in A'' \leftrightarrow B''$ do
        for $a' \in \tilde{A}''$ do
            for $b' \in \tilde{B}''$ do
                $sum := sum + a' * b'$;
            end do
        end do
    end do end do

In order to produce a high-level plan from the join surfaces and the hierarchies that will perform the computation in the manner suggested by the pseudocode, we need to reconcile the hierarchies with the π's and σ's that need to be performed at each let. We will discuss this next.

15.4.2 Reconciling the storage

In order to perform the appropriate partitioning, the pseudocode contains statements of the form,

$$\text{let } R' = \{\langle f', \tilde{R}'\rangle | f' \in \pi_f R \wedge \tilde{R}' = \pi_{\neg f}\sigma_{f=f'} R\};$$

This is just a formal way of saying that the join scheduler must figure out how to compute,

- $\pi_f R$, and
- $\pi_{\neg f}\sigma_{f=f'} R$, for each $i \in \pi_f R$.

It may be the case that the join scheduler can avoid computing these values explicitly if they can be enumerated directly from the given hierarchy. We need to develop a test to identity such situations. In our example, the order in which the fields are nested in the pseudocode is consistent with the order in which the fields appear in the hierarchy for A, so perhaps A' and A'' do not have to be explicitly computed.

Notice however that the fields in the hierarchy for B are not consistent with the nesting ordering. In this case, we are forced to explicitly compute B' and B''. The job of the join scheduler is to figure out how to do so efficiently. There are two basic approached to this. The first is to compute each of the partitions on the fly. In the case of B, this means that the join scheduler selects a storage format for B' and B'' and uses the available hierarchy to initialize each. This approach is particularly suitable for A, if A' and A'' must be computed. Since the field order in the hierarchy of A is consistent with what is required to compute A' and A'', it is likely that they can be computed efficiently on the fly using a small amount of additional storage.

But, consider B: its hierarchy is not consistent with what is required to compute B' and B''. For instance, if B' is to be computed using the $j \rightarrow i$ nesting implied by the hierarchy of indices, then all of the non-zero entries of B will have to be accessed in order to compute B'. Additional computation will be required to compute each occurrence of B''. In this case, it makes more sense to entirely remap B into a storage format from which B' and B'' can be computed directly. Thus, instead of incurring a large overhead for computing B' and B'' on the fly, by remapping B, in a sense, we compute B' and each of the B''s all at once. In a case like B, this will prove to be considerably more efficient than incrementally computing B and B'' on the fly.

Although there is clearly a tradeoff between these two strategies, in this thesis, we will only describe the second approach. The reason is that the first approach, although allowing some amount of the hierarchy to be exploited in some cases, can perform very poorly in others. The second approach, although less efficient in some cases, will be shown to only impose a cost proportional to the number of non-zero entries in the target matrix.

To recap: in order to perform join scheduling, we need

- a test for determining whether or not the partitions can be computed implicitly by directly enumerating the given hierarchy of indices, and
- a storage format that can be partitioned implicitly and an approach for remapping matrices with problematic hierarchies of indices to it.

15.4.3 Testing for consistency

In order to avoid computing partitions explicitly, it must be possible to compute the π's and σ's directly from the hierarchy of indices for a relation. This means that

- Fields must appear in the hierarchy in the same order in which they appear in the nesting, and
- The terms of the hierarchy are amenable to computing each of the π's and σ's.

If both of these properties can be shown to be true, then we will shown how the hierarchy of indices can be used to directly provide the partitions.

The top-level structure of a routine that testings a hierarchy against a nesting is shown in Figure 15.1.

15.4.4 Nesting ordering

In order to test that the fields of the hierarchy are in the same order as the nesting, we obviously need to ensure that order in which the fields appear in the hierarchy is that same as the order in which they appear in the nesting. In the example above,

$$\begin{aligned} \text{nesting} :& i \to j \to v \\ A :& i \to j \to v \\ B :& j \to i \to v, \end{aligned}$$

```
consistent? := function PartitionedHierarchy(R, hierarchy, v)
   for k := 1 to rank(H) do
      if any fields of R appear in v_k then
         term := car(hierarchy);
         if ¬(SameNesting(R, term, v_k)∧
                  AmmenableTerm(term)) then
            return #f;
         end if
      end if
      hierarchy := cdr(hierarchy);
   end do
   return #t;
end function
```

Figure 15.1: Testing a hierarchy for consistency with a nesting

it is clear that A satisfies this require and B does not. However, there are several additional properties that needs to be checked.

Suppose that, instead of being stored in CRS format, A is stored in the BlockSolve inode storage format. In this case, one of A's possible hierarchies is,

$$A : \{A.inode\} \rightarrow \{A.ii, A.i\} \rightarrow \{A.jj, A.j\} \rightarrow \{A.v\}$$

Although the order in which i, j, and v appear within the hierarchy is consistent with the nesting, this hierarchy of indices cannot be used to be used to directly provide the partitions.

The problem is with the $\{A.inode\}$ term. In order to compute $\pi_i A$, it would be necessary to traverse both of the $\{A.inode\}$ and $\{A.ii, A.i\}$ terms. More precisely, we need to ensure, at run-time, that no value of i appears within two different inodes. It is certainly possible to do so, and to compute $\pi_i A$ on the fly, but we have already ruled that approach out. Thus, instead of treating such terms as "unjoinable", as we did in Chapter 11, and instead of using them to compute $\pi_i A$ on the fly, we will deem the hierarchy to be inconsistent with the nesting.

There is another case that we need to test for, and that is when two fields from the same relation appear within the same join surface. This can happen, for instance, when only the diagonal of a matrix is accessed. Consider the following dense specification,

for $k := 1$ to n do
    $sum := sum + A[k, k]$
end do

In this case, the linear framework will construct a parametric equation in a single variable, t_1,

$$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} t_1 = \begin{pmatrix} I.k \\ A.i \\ A.j \end{pmatrix}$$

Notice that both $A.i$ and $A.j$ appear within the join surface for t_1. Because the join terms in the hierarchy are restricted to containing only one join field, there is no hierarchy that can be consistent with this nesting.

The code to determine that each term of a hierarchy appears in an order that is consistent with the nesting is shown in Figure 15.2. It ensures that the term under consideration exactly match the fields in the current level of the nesting.

$consistent?$:= function $SameNesting(R, term, \mathbf{v}_k)$
    $consistent? := (VarsOf(\mathbf{v}_k) \cap JoinIndices(R) = JoinIndices(R))$
end function

Figure 15.2: Testing term ordering

15.4.5 Amenable terms

Even if the order of the terms within a hierarchy of indices is consistent with a nesting, it does not follow that the partitions can be obtained directly from the hierarchy. If any of the terms within the hierarchy is not amenable to computing the appropriate π's and σ's, then the hierarchy cannot be used directly.

Suppose that A is stored in compressed-compressed row storage (CCRS) format. In addition to compressing the rows of A, this format compressed the row *indices* as well. This is advantageous when a sparse matrix has a high percentage of rows that are entirely zeros. One hierarchy of indices for this format is,

$$A : \{A.ii, A.i\} \rightarrow \{A.jj, A.j\} \rightarrow \{A.v\}$$

Note the addition of the $A.ii$ field, which reflects the compression of the row indices, i. Consider the first term of this hierarchy of indices, $\{A.ii, A.i\}$. Is this term amenable for computing the following expressions?

$$i' \in \pi_i A \quad \pi_{j,a}\sigma_{i=i'}A$$

Computing $i' \in \pi_i A$ involves, not only collecting all values of i, but also collecting all of the ii's corresponding to each value of i. This is required in order to compute $\pi_{j,a}\sigma_{i=i'}A$.

The problem occurs when nothing is known about the values produced by the term $\{A.ii, A.i\}$. If, for instance, there are multiple values of ii corresponding to a single i, then these values will have to be collected on the fly. Note, even if the black-box protocol specifies that A does not contain multiply entries for the same index (i.e., A combining operator is "no duplicates"), it does not follow that there cannot be multiple "rows" with the same index, i.[1] Since, we have already ruled out computing the π's and σ's on the fly, we will only be able to consider a term amenable if is can be shown that it can only produce *single* instances of the join field. In this case, computing the π's and σ's can be done by directly enumerating the results produced by the term. Without extending the black-box protocol, the term $\{A.ii, A.i\}$ cannot be shown to have this property, therefore, it must be not be considered amenable.

[1]Although, if there were, then these "rows" would have to have disjoint indices for j.

But what terms are considered amenable? Given the existing implementation of the black-box protocol described in Appendix C and the existing mechanism for assigning access methods to terms described in Section 9.3, the following are all of the terms that are amenable to directly providing the π's and σ's for partitioning R on the field f,

- The singleton lookup term, $\{R.f\}^1_{\texttt{lookup\_f}}$.
- The simple enum term, $\{R.f\}^*_{\texttt{enum\_f}}$, when the access method `enum_f` returns a stream of type `am_range_type`.
- The searching enum term, $\{R.ff, R.f\}^*_{\texttt{enum\_ff, lookup\_f, search\_f}}$.

It is fairly straightforward to seen why the first two terms are amenable. In the first case, a lookup term produces no more than one value, so multiple instances of i are not possible. In the second case, if the access method `enum_f` produces stream of type, `am_range_type`, as described in Section 14.3.1, then a simple `for` can be used to enumerate the values of i. This cannot result in multiple instances of i.

The third case, the searching enum term, is not so obvious. Why should a searching enum term, like $\{R.ff, R.f\}^*_{\texttt{enum\_ff, lookup\_f, search\_f}}$, be amenable but an indexing enum term, like $\{R.ff, R.f\}^*_{\texttt{enum\_ff, lookup\_f}}$, not? The reason is because of a quirk in the how the searching enum term is defined. Examining the definition in Section 9.3, it can be seen that the `search_f` method is required to be a singleton method. This means that no more than one value of ii will be returned from a search for a particular value if i. Thus, even if we enumerate the values of i using the `enum_ff` and `lookup_f` methods, the existence of the method `search_f` method guarantees that multiple ii's for a single i will not be encountered.

The code for testing the amenability of a term is shown in Figure 15.3.

Providing the partitions. In remains to be seen that, given a hierarchy that is consistent with a nesting, the appropriate partitions of the relation can be enumerated directly. This can be seen by examining each term of the hierarchy and its corresponding field in the nesting.

- First, consider the last term of the hierarchy, which corresponds to the value field of the relation. These are the only kinds of unjoinable terms allowed using the new partitioning rule. Since the only operation

$ammenable? :=$ function $AmmenableTerm(term)$
    match $term$ with
        pattern $\{R.f\}^1_{\texttt{lookup\_f}} \rightarrow$
            $ammenable? := \#t;$
        end pattern
        pattern $\{R.f\}^*_{\texttt{enum\_f}} \rightarrow$
            $ammenable? := (ResultType(\texttt{enum\_f}) = \texttt{am\_range\_type});$
        end pattern
        pattern $\{R.ff, R.f\}^*_{\texttt{enum\_ff}, \texttt{lookup\_f}, \texttt{search\_f}} \rightarrow$
            $ammenable? := \#t;$
        end pattern
        pattern $\_ \rightarrow$
            $ammenable? := \#f;$
        end pattern
    end match
end function

Figure 15.3: Testing the amenability of a term

required is to enumerate all of the instances of the value field, this can easily be done for all kinds of terms.

- Second, consider a join term for the relation R and its corresponding join field in the nesting, f. Suppose that *term* is such a join term and *hierarchy'* is the portion of the R's hierarchy that appears after *term*. The partitioning rule requires that the following expressions be directly computed:

$$f' \in \pi_f R \qquad \pi_{\neg f} \sigma_{f=f'} R$$

There are exactly three kinds of terms that are considered amenable to computing these expressions, For each of these terms, the set of $f' \in \pi_f R$ is easily computed:

  - $\{R.f\}^1_{\texttt{lookup\_f}}$. In this case, invoking the `lookup_f` method will produce the single value of f'. Because this method produces a singleton, there can be no duplicates.
  - $\{R.f\}^*_{\texttt{enum\_f}}$, when the access method `enum_f` returns a stream of type `am_range_type`. A **for** loop can be used to produce $f' \in \pi_i R$, because a **for** loop can be used to enumerate the contents of this stream without duplicates.
  - $\{R.ff, R.f\}^*_{\texttt{enum\_ff},\, \texttt{lookup\_f},\, \texttt{search\_f}}$. A loop over the values of ff using the `enum_ff` method, followed by an invocation of method `lookup_f` will produce all of the values of f within R. Because of the existence of the `search_f` singleton method, there can be no duplicates.

The only remaining point is how to obtain $\pi_{\neg i} \sigma_{f=f'} R$. Since it has been shown that the values of f can be computed without duplicates, the values of $\pi_{\neg i} \sigma_{f=f'} R$ can be obtained simply by binding f to f' and enumerating the tuples contained in, *hierarchy'*, the remaining portion of R's hierarchy.

f' can then be used to compute the corresponding t_k using, $Calc\_t_k$, which was described in Chapter 12.

- Examining the structure of the function *SameNesting*, it is evident that there cannot be any other cases.

To summarize: once a hierarchy of indices has been shown to be consistent with a nesting, the various partitions required by the new partitioning rule can be obtained simply by traversing hierarchy in the usual manner. No further computation needs to be performed on the fly and no addition storage is required.

15.4.6 Remapping inconsistent storage

If a hierarchy of indices fails the consistency test, it cannot be used to provide the necessary π's and σ's implicitly. Thus, they need to be computed explicitly.

Suppose that an inconsistent relation, R, is to be remapped to a consistent relation, R'. What should the schema of R' be? The obvious answer to use the index fields and value field of R as the schema for R'. However, this will not work in some cases. Consider again the example in which a matrix A is accessed only along its main diagonal,

```
for k := 1 to n do
    sum := sum + A[k, k]
end do
```

Because both $A.i$ and $A.j$ appear in the single join surface that is generated for this code, remapping to a relation, such as $A'(i, j, v)$ will still not provide a consistent relation!

Fundamentally, when a relation is to be remapped, it must be remapped so that it is consistent with the join surfaces, $t_1, \ldots, t_n$. In this case, it makes sense to just use these t_k's as the schema of the new relation. In the example above, this would mean remapping to the relation, $A'(t, v)$, which stores only the main diagonal of A. So, the task of remapping consists of computing the appropriate partitions of the parametric variables using the fields of the original relation and storing the results in a manner that can be easily accessed by the high-level plan.

In order to do this, we must identify a class of such storage formats that can serve as targets of remapping. Also, we need to specify how the remapping is to occur.

Storage formats for remapping. A data structure that consists of a sequence of nested hash tables, or similar dynamic data structures, can serve as the target of remapping. Such a data structure would have to have a level for each level of partitioning that needs to be computed. Assuming that the fields of original relation, R, appear in every join surface, then the target relation, R' will have the schema, $\langle t_1, \ldots, t_n, v\rangle$, and the hierarchy of indices for the target storage format will be,

$$\begin{aligned}&\{R.tt_1, R.t_1\}^*_{\texttt{enum\_tt}_1,\, \texttt{lookup\_t}_1,\, \texttt{search\_t}_1} \rightarrow \cdots\\ &\rightarrow \{R.tt_n, R.t_n\}^*_{\texttt{enum\_tt}_\texttt{n},\, \texttt{lookup\_t}_\texttt{n},\, \texttt{search\_t}_\texttt{n}}\\ &\rightarrow \{R.vv, R.v\}^*_{\texttt{enum\_vv},\, \texttt{lookup\_v}}.\end{aligned}$$

Since a searching enum term is specified to produce each of the t_i's, and since the order of the t_i's is consistent with the nesting order by construction, this hierarchy is consistent with the nesting.

There are two remaining details,

- If R is involved in only a subset of the joins in the high-level plan, then only those parametric variables will appear as fields in R'.
- Instead of copying the value $R.v$ to $R'.v$, each instance of $R'.v$ should refer to the same location as its corresponding entry of $R.v$. This can be accomplished by using the $R'.vv$ field to store a point to the corresponding $R.v$ and having the `lookup_v` method perform a pointer dereference.

Performing the remapping. Once the target storage format has been selected for R', code must be generated to initialize it with the appropriate values of R. In order to do this, the entries of R are enumerated using the existing hierarchy of indices and corresponding entries are inserted into R'. Since we already have the join surfaces that produce the t_k's and a hierarchy of indices for accessing R, and since this computation can be trivially phrased as a inner join query, the join scheduler and join implementer discussed in Chapters 11 and 12 can be used to generate this code. There are a few details that remain,

- If R is involved in only a subset of the joins in the high-level plan, then FME can be used to produce a restricted set of join surfaces in terms of the parametric variables of those joins.

- Having each $R'.v$ refer to the same location as its corresponding entry of $R.v$ eliminates the need to generate code to copy the values from R' back to R.

Introducing remapping. The mechanism for introducing the code to perform the remapping of the inconsistent sparse matrices is straightforward: if a sparse matrix is consistent with the nesting order, then it is used without remapping; otherwise, code to perform the remapping is generated along with a new, consistent hierarchy of indices, and the join surfaces are updated to refer to the new storage format. The code to perform this task is shown in Figure 15.4.

15.5 The join scheduler

After remapping has been performed, the hierarchies of indices for each of the relations appearing in the query are nicely aligned with the nesting order specified by the join surfaces of $\mathbf{H}$. This makes the task of $JoinScheduling$ very easy. All that is required is,

1. Form $\mathbf{H}$, $\mathbf{v}$, and $\bar{\mathbf{v}}$.

2. Form the original hierarchies of indices.

3. Remap inconsistent relations.

4. Generate a *Join* stage for each of the k join surfaces.

5. Generate $Unjoinable$ stages for each of the value fields.

The non-deterministic algorithm for join scheduling of general queries is shown in Figure 15.5.

15.6 The join implementer

Several changes must be made to the join implementation algorithm described in Chapter 12 in order to be able to transform the high-level plans

```
⟨new_hierarchies, new_H, new_v, new_v̄, code⟩ :=
    function Remap(hierarchies, H, v, v̄)
  new_hierarchies := ⟨⟩;
  code := [NOP];
  ⟨new_H, new_v, new_v̄⟩ := ⟨H, v, v̄⟩;
  while hierarchies ≠ ⟨⟩ do
    hierarchy := car(hierarchies);
    hierarchies := cdr(hierarchies);
    R := RelationsOf(hierarchy);
    if PartitionedHierarchy(R, hierarchy, v) then
      hierarchy' := hierarchy;
      code' := [NOP];
    else
      hierarchy' := generate the hierarchy of R';
      code' := generate code to initialize R';
      update new_H,new_v,new_v̄ to refer to R';
    end if
    new_hierarchies := hierarchy' :: new_hierarchies;
    code := code :: code';
  end match
end function
```

Figure 15.4: Remapping the inconsistent relations

$remap\_code, plan :=$ function $JoinScheduling(query)$
    $stage := 1$; $plan := \epsilon$; $i := 1$;
    -- *1. Form* $\mathbf{H}$, $\mathbf{v}$, *and* $\bar{\mathbf{v}}$.
    $\mathbf{H}, \mathbf{v}, \bar{\mathbf{v}} := FormParametricEquation(query)$;
    $\mathbf{P}, \mathbf{Q} := FindPandQ(\mathbf{H})$;
    $\mathbf{H} := \mathbf{PHQ}$;
    $(m, n) := size(\mathbf{H})$;
    -- *2. Form the original hierarchies of indices.*
    for $R \in RelationsOf(query)$ do
        $hierarchy_R := FindHierarchy(R)$;
    end do
    -- *3. Remap inconsistent relations.*
    $\langle \{hierarchy_{R'}\}, \mathbf{H}, \mathbf{v}, \bar{\mathbf{v}}, remap\_code \rangle :=$
            $Remap(\{hierarchy_R\}, \mathbf{H}, \mathbf{v}, \bar{\mathbf{v}})$;
    -- *4. Generate a* $Join$ *stage for each of the* k *join surfaces.*
    for $k := 1$ to n do
        -- *Extract the join surface for this join ...*
        $\mathbf{H}_k := \mathbf{H}(i : i + |group_k| - 1, 1 : k - 1)$;
        $\mathbf{h}_k := \mathbf{H}(i : i + |group_k| - 1, k)$;
        $\mathbf{v}_k := \mathbf{v}(i : i + |group_k| - 1)$; $\bar{\mathbf{v}}_k := \bar{\mathbf{v}}(i : i + |group| - 1)$;
        -- *Select the join terms of this stage ...*
        $join\_terms := \phi$;
        for $R'.f \in VarsOf(\mathbf{v}_k)$ do
            $term := car(hierarchy_{R'})$;
            $hierarchy_{R'} := cdr(hierarchy_{R'})$;
            $join\_term := join\_term \cup \{term\}$
        end do
        $step := Join(\mathbf{v}_k = \mathbf{H}_k \hat{\mathbf{t}}_k + \mathbf{h}_k t_k + \bar{\mathbf{v}}_k, bounds_{t_k}, join\_terms)$;
        $plan := plan +\!\!+ \langle step \rangle$;
        $stage := stage + 1$;
    end do

Figure 15.5: Join scheduling for general queries

Figure 15.5: (Continued)

```
-- 5. Generate Unjoinable stages for each of the value fields.
for hierarchy_R' ∈ {hierarchy_R'} do
    -- There is only one term remaining in hierarchy_R'
    term := car(hierarchy_R');
    step := Unjoinable(term);
    plan := plan ++⟨step⟩;
    stage := stage + 1;
end do
end function
```

for evaluating general queries into efficient low-level plans. The previous algorithm worked for queries in which all of the array references were strong, and, as a result, the only join operator used was $\bowtie$. In general queries, neither is true. Thus, the changes made are to account for the fact that some references are weak, which can result in ω's being generated for the value fields of some matrices, and that some joins will be implemented using $\leftrightarrow$ and $\rightarrow$.

15.6.1 Top-level

The top-level structure of the join implementer must be changed to account for the fact that code must be generated, not only for the case when a particular index is stored in a relation, but possibly also when it is not. Consider the following code in which the matrix, A, is stored in the CCRS format,

```
for i := 1 to n do
    for i := 1 to n do
        sum := sum + f(A[i, j]);
    end do
end do
```

If f is known to be side-effect free, but its behavior when $A[i,j] = 0$ is unknown, then it is safe to execute the iterations of the loop in any order, but every iteration must be executed. Thus, the sparsity guard computed for this loop is $SP = \#t$. The query used to represent this sparse computation will be,

for $\langle i, j, A.v\rangle \in I(i) \to A(i,v)$ do
  $sum := sum + f(RVAL(A.v));$
end do

and a high-level plan that might be generated for this query is,

Stage	Linear	A
1.	$Join(\mathbf{v}'_1 = \mathbf{h}'_1 t_1 + \cdots,$	$\{\{A.i, A.ii\},\})$
1.	$Join(\mathbf{v}'_2 = \mathbf{h}'_2 t_2 + \cdots,$	$\{\{A.j, A.jj\},\})$
2.	$Unjoinable($	$\{A.v\}\quad)$

The low-level plan generated from this schedule must account for two different cases,

- if $A[i,j]$ is stored in A, then its value must be produced, and
- if $A[i,j]$ is not stored in A, then the implicit value of 0 must be produced.

If $A[i,j]$ is a strong reference, then the compiler can use the fact that the computation need not be performed when $A[i,j]$ is not stored in order to avoid generating code for that case.

These are two basic approaches that the join implementer might use to handling the two cases of $A[i,j]$ being stored or not. The first, called the *run-time validation* approach, uses flags to record the validity of accesses to A and conditionals to ensure that only valid accesses are made. This is illustrated by the following low-level plan,

for $i := 1$ to n do
  -- $A_{\mathtt{search\_i}}[i]$.
  if $A_{\mathtt{search\_i}}(i)$ then
    $ii := A_{\mathtt{search\_i}}[i];\ valid_{ii} := \#t;$
  else

```
        valid_ii := #f;
    end if
    for j := 1 to n do
        -- A_search_j[ii, j].
        if valid_ii ∧ A_search_j(ii, j) then
            jj := A_search_j[ii, j]; valid_jj := #t;
        else
            valid_jj := #f;
        end if
        -- A_lookup_v[ii, jj].
        if valid_jj ∧ A_lookup_v(ii, jj) then
            t := A_lookup_v[ii, jj];
        else
            t := 0;
        end if
        -- The body.
        sum := sum + f(t);
    end do
end do
```

In this case, a temporary, t is assigned the result of the search, and $RVAL(A.v)$ is replaced with t to produce the body.

In the second approach, called the *decision tree* approach, a sequence of nested conditionals are used to ensure that the validity of each access method is tested only once. Copies of the query body that have been specialized for the particular context are placed at the leaves of this decisions tree. That is, a copy of the body placed in the context when all of the access methods are valid can use these methods to access the entry required by the body. A copy placed in a context when any of the access methods are invalid will have to use the default value, 0, for each entry required. This is illustrated by the following low-level plan,

```
for i := 1 to n do
    if A_search_i(i) then
        ii := A_search_i[i];
        for j := 1 to n do
            if A_search_j(ii, j) then
```

```
            jj := A_search_i[ii, j];
            if A_lookup_v(ii, jj) then
                -- A_search_i(i) ∧ A_search_j(ii, j) ∧ A_lookup_v(ii, jj)
                sum := sum + f(A_lookup_v[ii, jj]);
            else
                -- A_search_i(i) ∧ A_search_j(ii, j) ∧ ¬A_lookup_v(ii, jj)
                sum := sum + f(0);
            end if
        else
            -- A_search_i(i) ∧ ¬A_search_j(ii, j)
            sum := sum + f(0);
        end if
    end do
else
    for j := 1 to n do
        -- ¬A_search_i(i)
        sum := sum + f(0);
    end do
end if
end do
```

There are tradeoffs between the two approaches. For instance, when scheduling multistage plans, the run-time validation approach introduces many run-time checks that are not required by the decision tree approach. On the other hand, the copies of the body and conditionals created by the decision tree approach can result in code explosion. A third approach might be to use the run-time validation approach for the outer stage of the high-level plan, in which the run-time checks are performed infrequently, and to use the decision tree approach for the inner stages, in which fewer run-time checks will result in less overhead in the inner loops. Since we have not yet implemented either of these techniques, we do not have a feel for how the different approaches actually perform in practice.

15.6.2 Selecting a join operator

Suppose that the following step appears at stage k of the high-level plan.

$$Join(\mathbf{v}'_1 = \mathbf{h}'_1 t_1 + \bar{\mathbf{v}}'_1, bounds_{t_k}, \{term_A, term_B, term_C\})$$

Since the original query was,

$$I \rightarrow \ldots \rightarrow A \rightarrow \ldots \rightarrow B \rightarrow \ldots \rightarrow C \rightarrow \ldots$$

and since the partitioning rule given in Theorem 15.1 was used to produce the produce the high-level plan, it is safe to produce a low-level plan that evaluate the following query for this stage,

$$T \rightarrow \tilde{A} \rightarrow \tilde{B} \rightarrow \tilde{C}$$

where $A.f_A$, $B.f_B$, and $C.f_C$ are the join fields for this stage and

$$\begin{aligned}
T = \{\langle t_k, f_A, f_B, f_C\rangle | t_k \in bounds_{t_k} \wedge [\mathbf{H}'_k]_{A.f_A}\hat{\mathbf{t}}_k + [\mathbf{h}'_k]_{A.f_A} t_k + [\bar{\mathbf{v}}'_k]_{A.f_A} \\
\wedge [\mathbf{H}'_k]_{B.f_B}\hat{\mathbf{t}}_k + [\mathbf{h}'_k]_{B.f_B} t_k + [\bar{\mathbf{v}}'_k]_{B.f_B} \wedge [\mathbf{H}'_k]_{C.f_C}\hat{\mathbf{t}}_k + [\mathbf{h}'_k]_{C.f_C} t_k + [\bar{\mathbf{v}}'_k]_{C.f_C}\} \\
\tilde{A} = \{\langle f_A, \ldots\rangle \in \text{values produced by } term_A\} \\
\tilde{B} = \{\langle f_B, \ldots\rangle \in \text{values produced by } term_B\} \\
\tilde{C} = \{\langle f_C, \ldots\rangle \in \text{values produced by } term_C\}
\end{aligned}$$

This query can be evaluated using the following operations,

1. Enumerate the values of $t_k \in bounds_{t_k}$.
2. Compute the appropriate values of f_A, f_B, and f_C.
3. Searching A for the entries associated for f_A.
4. Searching B for the entries associated for f_B.
5. Searching C for the entries associated for f_C.

However, if this approach is taken for all stages then the low-level plan will enumerate all iterations of the original dense specification!

Suppose that it could be shown that the predicate $A \vee B$ dominates the query predicate; that is, $SP \Rightarrow A \vee B$. In this case, an iteration, t_k, does not have to be performed unless the corresponding entries exist in either A or B. The entries in the set "$A \vee B$" can be produced using the $\leftrightarrow$, so we can evaluate the following query instead of the previous,

$$((\tilde{A} \leftrightarrow \tilde{B}) \bowtie T) \rightarrow C.$$

This query can be evaluated using the following operations,

1. Perform an outer join on A and B on t_k.
2. Check that each t_k falls within $bounds_{t_k}$.
3. Searching C for the entries associated for t_k.

This new strategy will likely require far fewer iterations to be performed, since the number of t_k's in $\tilde{A} \leftrightarrow \tilde{B}$ is likely to be much smaller than in $bound_{t_k}$.

Similarly, if it can be shown that $A \wedge B$ dominates the sparsity guard then $\tilde{A} \bowtie \tilde{B}$ can be used to produce the relevant t_k's. Furthermore, it is preferable to use $\bowtie$ instead of $\leftrightarrow$, since it will likely produce fewer iterations. Thus, a good strategy for assigning join operators to each join stage is,

- Find the largest disjunctive predicate, $P_\vee$, whose terms are relations that appear in the stage, and which dominates the sparsity guard.
- Find the similarly largest conjunctive predicate, $P_\wedge$.
- Select the operator based upon,
  - If $|P_\wedge| \geq |P_\vee|$ then, the operator is $\bowtie$.
  - If $|P_\wedge| < |P_\vee|$ then, the operator is $\leftrightarrow$.
  - Otherwise, the operator is $\rightarrow$.

15.6.3 Join implementations

Although a complete catalog of join implementations is beyond the scope of this thesis, we will give examples of these here. We will assume that the joins being implemented are performed between the two fields, $A.f$ and $B.g$, using the terms,

$$term_A = \{A.f\}^*_{\texttt{enum\_f}} \qquad term_B = \{B.g\}^*_{\texttt{enum\_g}}$$

In Chapter 12, we discussed three basic strategies for implementing the $\bowtie$ join operator,

- Enumerate and Select,
- Sort and Merge, and
- Blocking

These three strategies can be used, with some modifications, to implement the $\leftrightarrow$ and $\rightarrow$ operators, as well.

15.6.4 Enumerate and Select

The basic form of this implementation for $\leftrightarrow$ operator is as follows,

```
for f_A ∈ A do
    if Calc_t_k(k, A.f, t̂_k, f_A) ∈ bounds_{t_k} then
        flag := #f;
        for g_B ∈ σ_{B.g=g_B} B do
            flag := #t;
            -- ⟨f_A, g_B⟩ ∈ (A ↔ B) available.
        end do
        if ¬flag then
            -- ⟨f_A, ω⟩ ∈ (A ↔ B) available.
        end if
    end if
end do
for g_B ∈ B do
    if Calc_t_k(k, B.g, t̂_k, g_B) ∈ bounds_{t_k} then
        if ∄f_A ∈ σ_{A.f=f_A} A do
            -- ⟨ω, g_B⟩ ∈ (A ↔ B) available.
        end if
    end if
end do
```

Notice that this implementation constrains two loop nests, one that computes $A \to B$, and one that computed $B \triangleright A$, which together compute $A \leftrightarrow B$. This basic implementation can be improved by index creation and the use of existing search methods.

15.6.5 Sort and Merge

Recall that the basic form of this strategy is,

A' := sort A on f;
B' := sort B on g;
for $\langle f_A, g_B \rangle \in merge_{\mathsf{op}}\ (A', B')$ do
  -- $\langle f_A, g_B \rangle \in (A$ *op* $B)$ *available.*
end do

Sorting is not affected by the choice of join operator; it is only $merge_{\text{op}}$ that changes. The key difference is that $merge_{\bowtie}$ skips entries that appear in only one of A or B, while $merge_{\leftrightarrow}$ and $merge_{\to}$ cannot. To illustrate this point, the implementation for $merge_{\leftrightarrow}$ is shown in Figure 15.6 for the case when the t_k's associated with $A.f$ and $B.g$ can be enumerated in increasing order.

15.6.6 Blocking

Extending the $\bowtie$ Blocking techniques to handle the $\leftrightarrow$ and $\to$ operators can be problematic. Consider for instance, the Blocked Nested Loop join implementation shown in Figure 12.14, which is based upon the following basic form,

for $A_i \subset A$ where $A = \biguplus_i A_i$ do
  for $B_j \subset B$ where $B = \biguplus_j B_i$ do
    for $\langle f_A, g_B \rangle \in (A_i \bowtie B_j)$ do
      ...
    end do
  end do
end do

```
declare $h_A$ : stream of $A'_{\texttt{enum\_f}}[\,]$; $h_A.init()$;
declare $h_B$ : stream of $B'_{\texttt{enum\_f}}[\,]$; $h_B.init()$;
while $h_A.valid() \vee h_B.valid()$ do

    -- $t_{kA}$ gets the next value, or $\infty$.
    if $h_A.valid()$ then
        $f_A := h_A.deref()$; $t_{kA} := Calc\_t_k(k, A.f, \hat{\mathbf{t}}_k, f_A)$;
    else
        $t_{kA} := \infty$;
    end if

    -- $t_{kB}$ gets the next value, or $\infty$.
    if $h_B.valid()$ then
        $g_B := h_B.deref()$; $t_{kb} := Calc\_t_k(k, B.g, \hat{\mathbf{t}}_k, g_B)$;
    else
        $t_{kB} := \infty$;
    end if

    $t_k := \min(t_{kA}, t_{kB})$;

    -- Load all entry from $A$ for $t_k$ into $b_A$.
    declare $b_A$ : bag;
    if $t_k = t_{kA}$ then
        while $h_A.valid() \wedge h_A.deref() = f_A$ do
            add $h_A.deref()$ to $b_A$;
            $h_A.incr()$;
        end do
    else
        add $\omega$ to $b_A$;
    end if
```

Figure 15.6: $merge_{\leftrightarrow}$

Figure 15.6: (Continued)

```
    -- Load all entry from $B$ for $t_k$ into $b_B$.
    declare $b_B$ : bag;
    if $t_k = t_{kB}$ then
        while $h_B.valid() \wedge h_B.deref() = g_B$ do
            add $h_B.deref()$ to $b_B$;
            $h_B.incr()$;
        end do
    else
        add $\omega$ to $b_B$;
    end if

    -- For $b_A \times b_B$ ...
    for $i_A := 1$ to $\#b_A$ do
        for $i_B := 1$ to $\#b_B$ do
            $\langle f_A, g_B \rangle = \langle b_A[i_A], b_B[i_B] \rangle$;
            -- $\langle f_A, g_B \rangle \in (A \leftrightarrow B)$ available.
        end do
    end do
end do
$h_A.close()$; $h_B.close()$;
```

This implementation can *not* be used for the $\leftrightarrow$ and $\rightarrow$ operators. This is because this implementation is based upon the $R = \biguplus_i R_i$ partitioning rule, which is not safe for $\leftrightarrow$ and $\rightarrow$. However, the Rough Hashing implementation based upon the following basic form,

```
for A.f ∈ A do
    add A.f to hashtbl_A;
end do
for B.f ∈ B do
    add B.f to hashtbl_B;
end do
for w_A := 1 to W do
    for ⟨A.f, B.g⟩ ∈ (hashtbl_A[w_A] op hashtbl_B[w_B]) do
        ...
    end do
end do
```

can be used because the buckets of the hash tables form a partition of A and B that is consistent with the general partitioning rule of Theorem 15.1.

15.7 An example

Consider the following dense specification,

```
for i := 1 to n do
    for i := 1 to n do
        C[i,j] := C[i,j] + A[i,j] + B[i,j];
    end do
end do
```

in which C is a dense matrix and A and B are sparse matrices stored in the CRS format. The query derived from this specification is,

for $\langle i, j, C.v, A.v, B.v\rangle \in$
$\sigma_{BVAL(A.v) \vee BVAL(B.v)}$
$(I(i,j) \rightarrow A(i,j,v) \rightarrow B(i,j,v))$ do

$LVAL(C.v) := RVAL(C.v) + RVAL(A.v) + RVAL(B.v);$
end do

Suppose that an $\mathbf{H}$ is constructed with the obvious join surfaces that yields and $i \to j$ nesting. In this case, the following hierarchies of indices are consistent with this nesting,

$$\{A.i\}^*_{\texttt{enum\_i}} \to \{A.jj, A.j\}^*_{\texttt{enum\_jj},\,\texttt{lookup\_j},\,\texttt{search\_j}} \to \{A.v\}^1_{\texttt{lookup\_v}}$$
$$\{B.i\}^*_{\texttt{enum\_i}} \to \{B.jj, B.j\}^*_{\texttt{enum\_jj},\,\texttt{lookup\_j},\,\texttt{search\_j}} \to \{B.v\}^1_{\texttt{lookup\_v}}$$
$$\{C.i\}^*_{\texttt{enum\_i}} \to \{C.j\}^*_{\texttt{enum\_j}} \to \{C.v\}^1_{\texttt{lookup\_v}}$$

Thus, no remapping needs to be done.

One high-level plan that might be derived by the join scheduler is,

Stage	Linear	C	A	B	
1.	$Join(\mathbf{v}'_1 = \mathbf{h}'_1 t_1 + \cdots,$	$\{\{C.i\},$	$\{A.i\}$	$\{B.i\}$	$\})$
2.	$Join(\mathbf{v}'_2 = \mathbf{h}'_2 t_2 + \cdots,$	$\{\{C.j\},$	$\{A.j, A.jj\}$	$\{B.j, B.jj\}$	$\})$
3.	$Unjoinable($	$\{C.v\}$			$)$
4.	$Unjoinable($		$\{A.v\}$		$)$
5.	$Unjoinable($			$\{B.v\}$	$)$

The next stage of compilation is join implementation. For each of the join stages, it can be shown that $BVAL(A.v) \vee BVAL(B.v) \vee BVAL(C.v)$ dominates the query predicate. However, since C is dense, $BVAL(C.v)$ is always true, so this simplifies to $BVAL(A.v) \vee BVAL(B.v)$. Thus, each of the join stages can be implemented using an outer join on the indices of A and B.

In order to perform join implementation, it must be specified whether the run-time validation or decision tree approach is being used and how each join is to be implemented. Suppose that the decision tree approach is used with a nested loop outer join implementation, then for each join stage the following code might result,

-- *Computes* $A \to B$.
for $i \in A_{\texttt{enum\_i}}[\,]$ do
    if $i \in B_{\texttt{enum\_i}}[\,]$ then
        for $jj_A \in A_{\texttt{enum\_jj}}[i]$ do

```
            j := A_lookup_j[i, jj_A];
            if B_search_j(i, j) then
                jj_B := B_search_j[i, j];
                C[i, j] := A_lookup_v[i, jj_A] + B_lookup_v[i, jj_B];
            else
                C[i, j] := A_lookup_v[i, jj_A] + 0;
            end if
        end do
        for jj_B ∈ B_enum_jj[i] do
            j := B_lookup_j[i, jj_B];
            if ¬A_search_j(i, j) then
                C[i, j] := 0 + B_lookup_v[i, jj_B];
            end if
        end do
    else
        for jj_A ∈ A_enum_jj[i] do
            j := A_lookup_j[i, jj_A];
            C[i, j] := A_lookup_v[i, jj_A] + 0;
        end do
    end if
end do
-- Computes B ▷ A.
for i ∈ B_enum_i[ ] do
    if i ∉ A_enum_i[ ] then
        for jj_B ∈ B_enum_jj[i] do
            j := B_lookup_j[i, jj_B];
            C[i, j] := 0 + B_lookup_v[i, jj_B];
        end do
    end if
end do
```

In the preceding code, we have omitted each of the $M_{\texttt{lookup\_x}}(\dots)$ tests in order to make the code more readable.

15.8 Future work

As we mentioned in the beginning of this chapter, none of the material described here for handling general queries has been implemented. Obviously, this is an important task that must be done. In the process of doing so, aspects of this material may be shown to be deficient. For a start, heuristics must be developed to guide these mechanisms, Furthermore, while it is impossible to anticipate exactly what else might be deficient, there are several areas of this work that need improvement.

15.8.1 Improvements on the methods

The remapping technique discussed in Section 15.4 is rather strict with respect to the types of hierarchies of indices that it will accept without wholescale remapping. There are several optimizations that might be made to these techniques.

One point that was not mentioned is this: it should be possible to accept a hierarchy as consistent if its innermost join term is not amenable, but the combining operator for the relation is "no duplicates". This is because duplicate indices cannot arise in the last term without having duplicate entries occuring within the relation. It is not clear that this situation occurs in practice, though.

We ruled out computing π's and σ's on the fly, but there are some obvious cases where it might be worth while to do so. One instance is when a hierarchy of indices is consistent with a nesting order, but one or more of its terms are not amenable. In this case, it should be possible to compute just these terms, as part of the high-level plan, without having to remap the entire relation. Of course, there is the other extreme of computing *everything* on the fly, but it is not clear that this is preferable to remapping to a more appropriate storage format. Experiments should be run to discover guiding principles that can be build into a heuristic to choose the best approach.

15.8.2 Deterministic scheduling

One of the nice properties of the join scheduling algorithm given in Chapter 11 is that all of the non-determinism of the algorithm was isolated in a single `chose` construct in Figure 11.1. This is not the case with the join scheduler presented in this chapter. The benefit of having a single point of non-determinacy is that it makes it very easy to insert a single heuristic to make the algorithm deterministic. In order to develop heuristics for the general join scheduling algorithm, we would have to first restructure the algorithm in Figure 15.5 in a manner similar to the original inner join scheduling algorithm.

15.8.3 Combining the conjunctive and general frameworks

We have offered the query optimization techniques of this chapter as an alternative to those presented in Chapters 11 and 12. Is it possible to merge these techniques into a hybrid query optimization method? The reason is that the techniques presented earlier produce much better results for queries containing $\bowtie$'s than those in this chapter. For a start, the join scheduler of Chapter 11 was able to exploit the natural partitions of the storage formats without the need for remapping. A hybrid technique would hopefully be able to provide the efficient schedules for $\bowtie$ operators while performing remapping only when necessary for scheduling $\leftrightarrow$ and $\rightarrow$ operators.

15.9 Related work

Because they are a more recent development, there has not been nearly as much work done on scheduling queries containing outer join as there has been for queries containing inner joins. Recent work in this area can be found in [90], [17], and [18]

Chapter 16

Fill and Annihilation

It is surprising how useful a sparse compiler that does not handle fill or annihilation can be. Libraries are available that not only discretize a problem and form a corresponding linear system but also store this system in a sparse matrix storage format appropriate for computation. In this situation, the numerical solution can often be computed without having to change the sparsity of the linear system. This is especially true when indirect, or iterative, solvers are used.

However, in order to handle more general codes, a sparse compiler must be able to handle the cases of *fill*, the creation of non-zero entries, and *annihilation*, the deletion of zero entries, within sparse matrices. Our existing compiler does not handle either, but in this chapter, we will present methods that can be used to extend this implementation to handle both.

16.1 The basics

Assume that a reference to a sparse matrix, C, appears in an l-value position in a sparse computation,

$C[\mathbf{F}_C i + \mathbf{f}_C] := rhs;$

In Chapter 8, it was stated that fill can occur in C when

$$\neg BVAL(C.v) \wedge NZ(rhs),$$

and annihilation can occur when

$$BVAL(C.v) \wedge \neg NZ(rhs).$$

In order to handle fill and annihilation, the compiler must perform three tasks,

- identify where it might occur,
- transform the query to account for these cases, and
- obtain code from the black-boxes that will perform the modifications to the storage formats.

The first was discussed in Chapter 8. The later two are discussed below.

16.2 Using dynamic data structures

A problem with many sparse matrix storage formats is that they do not provide for efficient insertion and deletion of non-zero entries. The CRS format described in Section 7.3.4 is an instance of this. Inserting or deleting a single entry from the middle of a *avalues* requires the reallocation of the entire vector.

What is often done in practice when modifications have to be made to the structure of these storage formats is to remap the entire sparse matrix to a dynamic data structure, like a hash table, to make a number of such modifications, and to reallocate an entirely new sparse matrix from the final entries in the dynamic data structure.

A sparse compiler can do something similar. Consider the following code, in which it has been determined that fill or annihilation can occur to C,

for $i := \ldots$ do
    $C[i,j] := rhs;$
end do

and whose corresponding query is,

for $\langle i, j, C.v, \ldots \rangle \in \ldots$ do
    $LVAL(C.v) := rhs;$
end do

The following modifications will have to be made to this query,

1. Code is inserted before and after the query to map C to and from a dynamic data structure C'. All references within the query to C are changed to C'.

2. An appropriate data structure with an appropriate set of access methods must be selected for C'.

3. The body of the loop must be updated in order to perform the insertion and deletion operations to C' when they are required.

Each of these points deserves further elaboration.

Unpacking and packing. We will call the conversion from a sparse matrix storage format to a dynamic data structure *unpacking* and the conversion back *packing*.

$unpack(C, C');$
for $\langle i, j, C.v, \ldots \rangle \in \ldots$ do
    $LVAL(C'.v) := rhs;$
end do
$pack(C', C);$

Unpacking a sparse matrix into a compiler selected data structure, such as a hash table, can be done in a straightforward manner. All that needs to be done is to have a query formed that copies the values form C to C', and then to use the existing query optimization techniques to schedule its efficient evaluation.

$$unpack(C, C'); \quad \longrightarrow \quad \begin{array}{l} init(C'); \\ \textsf{for } \langle i, j, C.v \rangle \in C \textsf{ do} \\ \quad insert(C', i, j, C.v); \\ \textsf{end do} \end{array}$$

Packing is not so straightforward. The difficulty is that the current version of the black-box protocol, while providing mechanisms for accessing a storage format, does not provide mechanisms for creating a storage format. One possible way of extending the protocol is to add a `create` method to every black-box. Such a method would take a C' as its argument and use its entries to reconstitute the storage format.

$$pack(C', C); \quad \longrightarrow \quad C_{\texttt{create}}[C'];$$

For the remainder of this chapter, we will assume that exactly such an extension has been made.

Since these are transformations on the query, they must be performed after query formulation and before query optimization.

Selecting access methods for C'. The choice of data structures for C' depends upon the type of access that is required by the join scheduler. For instance, if C does not appear in the query predicate, then the entries of C' will not be used to determine which iterations must be performed. In this case, a simple hash table that provides a single search method can be used,

$$C'_{\texttt{search\_v}}[i, j] \rightarrow v$$

However, if C appears in the query predicate then its entries must be enumerated in order to determine the iterations to be performed. Consider the following loop nest, which performs a matrix copy,

for $i := 1$ to n do

```
    for j := 1 to n do
        C[i,j] := A[i,j];
    end do
end do
```

If A and C are both sparse, then the query predicate for this code will be dominated by $BVAL(A.v) \vee BVAL(C.v)$. If the general query optimization techniques of Chapter 15 are used to schedule this query, then a data structure must be selected for C' that is consistent with the join surfaces selected to schedule the loop. This is because the remapping techniques described in Chapter 15 cannot be used for C, since they do not account for fill.

But since the join surfaces are not determined until join scheduling is under way, it is not until this point that the access methods necessary for C' can be known. Thus, dynamic data structures that are used to handle fill and annihilation, like C', must be treated specially during join scheduling. In particular, the exact access methods of C' must be left unspecified until **H** has been computed. The join scheduling algorithm in Figure 15.5 must be modified so that, while the hierarchies are being computed for the other sparse matrices during Step 2, the structure of **H** is used to derive consistent access methods for C'. Then, *JoinScheduling* can proceed normally.

Modify the body. Prior to running the join implementer, the body of the loop must be updated to invoke the appropriate insertion and deletion operations on C'. Each statement,

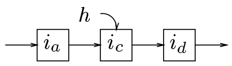
$$LVAL(C'.v) := rhs;$$

in the body will be replaced by,

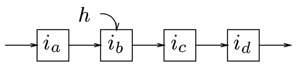
```
if NZ(rhs) then
    if BVAL(C'.v) then
        LVAL(C'.v) := rhs;
    else
        insert(C', i, j, rhs);
    end if
else if BVAL(C'.v) then
    delete(C', i, j);
```

and then join implementation can proceed normally. Note that this substitution ensures that $LVAL(C'.v)$ is never evaluated when $C.v$ could be ω.

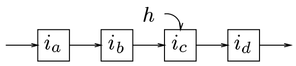
There is one final point that remains: if C appears in the query predicate, then the entries of C' are used to determine the iterations to be performed. Care must be taken when creating and deleting entries from C' that these operations do not incorrectly affect the iterations performed. Suppose for instance that the non-zero entries of C' are being enumerate with a stream whose handle is h.



Suppose that h points to the entry with index i_c when an entry for i_b is inserted. The insertion must be done so that h still points to i_c.



incorrect



correct

In order to ensure this, all active handles to entries of C' must be passed to the insertion and deletion operations for possible updating.

$insert(C', i, j, rhs, h_i, h_j)$;
$delete(C', i, j, h_i, h_j)$;

16.3 Optimizations

There are several important optimizations that can be made to the techniques described in the previous section.

Killing definitions. Suppose that a dense specification has the form,

```
for i := 1 to n do
    for j := 1 to n do
        C[i,j] := f(A[i,j]);
    end do
```

```
end do
```

If it can be determined by array region analysis ([121], [96], [104]) that all entries of C are overwritten by this loop, then there is no need to unpack these entries to C'. Instead, C' can simply be initialized and then used.

$init(C')$;
for $i := 1$ to n do
    for $j := 1$ to n do
        $insert(C', i, j, f(LVAL(A.v)))$;
    end do
end do
$C_{\texttt{create}}[C']$;

This optimization not only eliminates the overhead of copying entries from C to C', but it may simplify the query predicate as well. In the example above, the sparsity guard, and hence the query predicate, for the original loop was $BVAL(A.v) \vee BVAL(C.v)$, but the query predicate for the second loop is just $BVAL(A.v)$, since $BVAL(C.v)$ is always false.

In order for this optimization to take place, it must be shown that,

- all entries of C are written, and
- no entries of C are read.

A common occurrence of this situation is when the matrix C is being initialized with 0's:

for $i := 1$ to n do
    for $j := 1$ to n do
        $C[i, j] := 0$;
    end do
end do

Since the query predicate after this optimization is $\#f$, the query can safely be deleted and the only code remaining is,

$init(C')$;
$C_{\texttt{create}}[C']$;

Precomputing storage. Consider the following loop nest,

```
for t := 1 to 10 do
    for i := 1 to n do
        C[i] := A[i] * B[i];
    end do
    for i := 1 to n do
        A[i] := α * A[i];
    end do
    for i := 1 to n do
        B[i] := β * B[i];
    end do
end do
```

The techniques previously described would require that C be unpacked and packed in each iteration of the t loop. If it can be down that α and β are never 0, then this is unnecessary, since the sparsity of A and B will not change for the duration of the t loop. In order to prevent the extraneous unpacking and packing of C, the first iteration of t should be peeled ([121]) from the the loop and the query assertion

$$BVAL(A.v) \wedge BVAL(B.v) \Leftrightarrow BVAL(C.v)$$

should be added to the query predicates of the remaining iterations.

```
-- "Peeled" iteration. for i := 1 to n do
    C[i] := A[i] * B[i];
end do
for i := 1 to n do
    A[i] := α * A[i];
end do
for i := 1 to n do
    B[i] := β * B[i];
end do
-- Remaining loop nest. for t := 2 to 10 do
    for i := 1 to n do
        -- BVAL(A.v) ∧ BVAL(B.v) ⇔ BVAL(C.v).
```

```
            C[i] := A[i] * B[i];
        end do
        for i := 1 to n do
            A[i] := α * A[i];
        end do
        for i := 1 to n do
            B[i] := β * B[i];
        end do
    end do
```

After this transformation, a predicate must be added to the query predicate of the remaining loop nest to indicate that the reference to C is a strong reference. That way, the compiler will generate code to handle fill for the peeled iteration but not in the remaining loop nest.

In order for this optimization to be performed, it must be shown that,

- The first iteration of the loop can be safely peeled, and
- The sparsity of the matrix in question does not change after the first iteration of the loop.

An even more aggressive means of preallocating storage for a numerical computation is to execute the computation "symbolically". That is, a copy of the numerical computation is made and altered to compute an approximation of the final sparsity of a matrix by modeling the input matrices as bit matrices (i.e., $0 =$ zero, $1 =$ non-zero) and mapping the arithmetic operators to their conservative boolean equivalents (e.g., $+ \rightarrow \vee$, $* \rightarrow \wedge$). This method is closely related to the techniques of abstract interpretation ([42], [41]) and is widely used in sparse matrix factorization codes ([64], [99]).

16.4 Related work

Bik's sparse compiler ([19]) generates code that packs and unpacks single rows or columns of sparse matrices at a time. For instance, given a loop nest that assigns to a sparse matrix:

```
    for i := ... do
        for j := ... do
```

```
        C[i,j] := ... C[i,j] ... ;
    end do
end do
```

Bik's sparse compiler might generate code that accesses a single row of C at a time. In this case, access pattern expansion (Section 2.4.4) might be used to scatter an entire row of C into a dense vector, v, prior to performing all computations using v. Then, when the row is gathered from v back to C all occurrences of fill and annihilation can be handled at once.

```
for i := ... do
    -- Scatter C[i,*] to v.
    unpack(C[i,*], v);
    for j := ... do
        v[j] := ... v[j] ... ;
    end do
    -- Gather C[i,*] from v.
    pack(v, C[i,*])
end do
```

There are two advantages to this approach over our approach, which unpacks and packs the entire matrix. First, the references to v are simple array references, which are much more efficient to access than a hash table. Second, the insertions and deletions of non-zeros to each row of C are performed all at once, which requires less overhead than if they occured individually.

While Bik's methods handle general patterns of fill and annihilation, he does not consider preallocating storage. We believe this to be an important optimization in practice: witness the use of symbolic factorization in hand-written sparse direct methods. It is also extremely important for generating efficient access to communication buffers in message passing. For the sorts of codes we are primarily interested in, we believe that handling preallocation will give a greater performance gain than scattering and gathering single rows. However, we recognize that there are many cases where this is a better technique for handling fill and annihilation than hash tables.

16.5 An example

Consider the following dense specification for copying one sparse vector to another.

for $i := 1$ to n do
    $C[i] := A[i];$
end do

the query for which is,

for $\langle i, A.v, C.v\rangle \in \sigma_{BVAL(A.v) \vee BVAL(C.v)}(I(i) \to A(i,v) \to C(i,v))$ do
    $C[i] := A[i];$
end do

We will consider several alternatives for scheduling this code.

Basic method. The basic method for handling fill and annihilation can be used without any optimizations. In this case, an outer join on A and C is used to enumerate the appropriate iterations of the query. Suppose that the decision tree approach to join implementation is used and that the outer join is implemented using the sort and merge strategy. This is the code that might be generated:

-- *Unpack* C.
$init(C');$
for $ii \in C_{\texttt{enum\_ii}}[\,]$ do
    $insert(C', C_{\texttt{lookup\_i}}[ii], C_{\texttt{lookup\_v}}[ii], h_{C'});$
end do
-- *Initialize the streams.*
declare h_A : stream of $A_{\texttt{enum\_ii}}[\,];$
declare $h_{C'}$: stream of $C'_{\texttt{enum\_ii}}[\,];$
$h_A.init();\ h_{C'}.init();$
-- *While both streams are not empty* ...
while $h_A.valid() \wedge h_{C'}.valid()$ do
    $ii_A := h_A.deref();\ ii_{C'} := h_{C'}.deref();$
    $i_A := A_{\texttt{lookup\_i}}[ii_A];\ i_{C'} := C'_{\texttt{lookup\_i}}[h_{C'}.deref()];$
    if $i_{C'} = i_A$ then

```
        C'_lookup_v[ii_C'] := C'_lookup_v[ii_C']
        h_A.incr(); h_C'.incr();
    else if i_C' < i_A then
        delete(C', i_C', h_C'); h_C'.incr();
    else
        insert(C', i_A, A_lookup_v[ii_A], h_C'); h_A.incr();
    end if
end do
-- Insert the trailing elements of A into C.
while h_A.valid() do
    ii_A := h_A.deref(); i_A := A_lookup_i[ii_A];
    insert(C', i_A, A_lookup_v[ii_A], h_C'); h_A.incr();
end do
-- Delete the trailing elements from C.
while h_C'.valid() do
    ii_C' := h_C'.deref(); i_C' := C'_lookup_i[h_C'.deref()];
    delete(C', i_C', h_C'); h_C'.incr();
end do
h_A.close(); h_C'.close();
-- Pack C.
C_create[C']';
```

Recognizing killing definitions. Since the values of C are not used within the loop and are no longer available after the loop, this vector copy constitutes a killing definition of C. Thus, the values of C need not be copied to C', and then a left outer join may be performed between A and C.

```
-- Unpack C.
init(C');
-- Perform the computation.
for ii ∈ A_enum_ii[ ] do
    insert(C', A_lookup_i[ii], A_lookup_v[ii], h_C');
end do
-- Pack C.
C_create[C']';
```

Part IV

Conclusions

Chapter 17

Performance Results

In Chapter I, we argued that sparse compilation can be a useful tool for the numerical application developer. We also have argued that the relational model provides a natural framework in which to formulate the problem of sparse compilation, and that the sparse implementations obtained by our compiler are comparable to those written by hand or obtained by existing sparse compilers. In this chapter, we will substantiate these claims with concrete performance results obtained from a variety of applications and sparsity patterns.

Unless otherwise stated, the following are true of all of the results presented in this chapter and the rest of the thesis,

- Mflops were measured using the following formula,

$$\frac{2 * \text{Nzs}}{1,000,000 * \text{elapse time}}$$

  That is, we do not directly count the flops performed by the computation; we count the number of flops *needed* to perform the computation. If a particular storage format requires that many useless flops be performed, then its mflops performance will suffer.

- All runs were performed on a single thin node of the Cornell Theory Center SP-2. According to the online documentation ([98]), each of these nodes has the following characteristics,

  - Roughly equivalent to RS/6000 model 390.
  - 64 KB data cache.
  - 64 bit memory bus.
  - 1-4 GB disk.
  - 128-256 MB memory.

  Also, each of these nodes were dedicated to running batch jobs, and there were no other computationally intensive processes competing for resources on the same node during the timed runs.

- Before running the timed experiments, the computation was performed once, untimed. This has the effect of "warming" the cache with the data set. Only then were the timed experiments performed.
- The characteristics of each matrix, such as size, number of non-zeros, and goodness, can be found in Appendix A.
- The following vendor supplied compilers were used, with full optimization, to generate the test programs,
  - `xlc` (C) version 3.1.4.3.
  - `xlC` (C++) version 3.1.4.3.
  - `xlf` versions 3.2.4.2 and 4.1.0.3.

17.1 CRS CG & GMRES: Bernoulli vs. PETSc

We chose to compare the performance of code generated by our compiler with hand-written code by compare the performance of Krylov sparse solvers generated by our compiler with those found in the PETSc library([61, 10]). Care was taken to ensure that the compiler generated versions of these solvers produced the same numerical results as the hand-written versions. For these tests, all matrices were stored in CRS format.

Table 17.1 shows the performance results for the conjugate gradient solvers, and Table 17.2 shows the results for the GMRES solvers. These results indicate that the performance of the sparse solvers generated by the Bernoulli compiler is comparable with PETSc solvers, which were written by hand.

Table 17.1: Hand-written and compiler generated CG, in mflops

Grid	Mflops	
Name	Bernoulli	PETSc
1138_bus	26.339	20.874
662_bus	32.064	25.841
685_bus	31.989	24.776
bcsstm27	26.339	33.289
gr_30_30	24.849	21.912
nos4	33.168	23.128
nos5	28.528	23.359
nos6	32.625	25.101
nos7	25.137	21.931

Table 17.2: Hand-written and compiler generated GMRES, in mflops

Grid	Mflops	
Name	Bernoulli	PETSc
add20	30.739	34.696
add32	20.116	19.865
e05r0000	29.602	34.235
memplus	19.080	19.194
sherman1	33.107	36.087
medium	37.026	42.169
small	27.965	15.813
tiny	8.026	1.587

17.2 BlockSolve

Table 17.3 shows the performance of the MVM code from the BlockSolve library and code generated by our compiler, for 12 matrices. Each matrix was stored in the *clique/inode* storage format used by the BlockSolve library and was formed from a 3d grid with a 27 point stencil. These results indicate that the performance of the compiler-generated code is comparable with the hand-written code, even for as complex a data structure as BlockSolve storage format.

Table 17.3: Hand-written/Compiler-generated (Mflops)

Grids			Mflops	
d	n	c	Hand-written	Compiler-generated
3	10	1	4.16	4.65
3	10	3	16.43	17.55
3	10	5	23.03	24.23
3	10	7	28.04	28.89
3	17	1	4.15	4.40
3	17	3	16.24	17.32
3	17	5	23.52	24.26
3	17	7	26.21	27.00
3	25	1	4.22	4.40
3	25	3	16.19	17.14
3	25	5	22.85	23.05

17.3 CCS MVM: Bernoulli vs. Bik

Bik has provided a Web interface to his compiler ([29]). The user can fill out and submit a form describing the implementation that they want to have generated. The compiler will then be invoked with the appropriate source code, and the resulting output code will be presented back to the user. We used this server to generate code for performing MVM where the matrix is stored in the CCS format. The source file produced by Bik's server contained the following version information,

```
C This code was generated by MT1 Revision: 4.6
C at Tue Jan 28 18:02:13 PM MET 1997.
```

We had to make some hand-modifications to the generated code in order to make it fit within our experimental setup. Care was taken to ensure that these modifications did not affect performance. Table 17.4 compares the performance of MVM code generated by both of our compilers. Except for one case, the performance results are similar.

There is one minor difference between the two sparse implementations. The dense specification that served as input to the Bernoulli compiler was of the form,

for $j := 1$ to n do
  for $i := 1$ to n do
    $Y[i] := y[i] + A[i,j] * X[j]$
  end do
end do

while the dense specification that served as input to Bik's compiler was of the form,

for $j := 1$ to n do
  if $X[j] \neq 0$ then for $i := 1$ to n do
      $Y[i] := Y[i] + A[i,j] * X[j]$
    end do
  end if
end do

For most of the runs, this difference appears not to have resulted in a large difference in performance. The exception is the row marked with an $(*)$. In case, most of the entries in x are 0, which accounts for huge difference in performance.

Table 17.4: Bernoulli vs. Bik: MVM

Grids			Mflops	
d	n	c	Bernoulli	Bik
2	10	1	17.812	16.283
2	10	3	28.18	27.973
2	10	5	21.440	21.586
2	10	7	22.086	21.675
2	17	1	19.279	18.000
2	17	3	19.462	18.836
2	17	5	21.128	21.028
2	17	7	20.962	21.807
2	25	1	19.752	18.470
2	25	3	19.262	18.454
2	25	5	20.955	21.346
2	25	7	21.229	21.458
3	10	1	15.990	13.910
3	10	3	19.623	19.819
3	10	5	20.538	21.978
3	10	7	21.045	22.695
3	17	1	15.457	13.679
3	17	3	19.214	19.432
3	17	5	20.010	21.938
3	17	7	20.722	21.744
3	25	1	14.483	13.148
3	25	3	19.562	19.000
3	25	5	20.923	21.604
Name			Bernoulli	Bik
662_bus			18.504	15.063
685_bus			19.464	17.428
1138_bus			15.220	11.859
(∗)e05r0000			24.294	110.282
arco4			19.580	21.930

17.4 CCS MMM: Bernoulli vs. Bik

We also used Bik's server to generate code to perform sparse MMM, in which each matrix is stored in the CCS format. The source file produced by Bik's server contained the following version information,

```
C This code was generated by MT1 Revision: 4.6
C at Tue Jan 28 18:03:23 PM MET 1997.
```

Table 17.5 shows the results of $C = A * B$. The results for Bik's compiler are noticeably better because his compiler performs access pattern expansion to obtain efficient indexing of the sparse vectors in the inner loops. Since our current implementation of the join implementer is so primitive, we are only able to generate binary searches to resolve indices. A more sophisticated implementation of the join implementer would likely obtain performance comparable with his.

Table 17.5: Bernoulli vs. Bik: $C = A * B$

Grids			Mflops	
d	n	c	Bernoulli	Bik
2	10	1	1.674	5.243
2	17	1	1.604	4.758
2	25	1	1.545	4.828
3	10	1	1.174	5.282
3	17	1	1.121	5.283
3	25	1	1.105	5.229

The results of $C = A * B^T$, $C = A^T * B$, and $C = A^T * B^T$, are shown in Table 17.6, Table 17.7, and Table 17.8, respectively. In these cases, we perform noticeably better than Bik's compiler. The fundamental reason why occurs is because we preallocate C, and that our compiler was able to take advantage of this, and his compiler was not. Our compiler does not presently generate code to handle fill, so, before performing the computation, we must precompute what the sparsity for C will be and carefully allocated storage for this sparsity. Our compiler is able to use the fact that the reference to

Table 17.6: Bernoulli vs. Bik: $C = A * B^T$

Grids			Mflops	
d	n	c	Bernoulli	Bik
2	10	1	2.724	0.559
2	17	1	2.755	0.227
2	25	1	2.682	0.110
3	10	1	2.306	0.106
3	17	1	2.234	0.021
3	25	1	2.222	0.007

Table 17.7: Bernoulli vs. Bik: $C = A^T * B$

Grids			Mflops	
d	n	c	Bernoulli	Bik
2	10	1	1.615	0.193
2	17	1	1.506	0.072
2	25	1	1.455	0.034
3	10	1	1.114	0.028
3	17	1	1.055	0.006
3	25	1	1.041	0.002

Table 17.8: Bernoulli vs. Bik: $C = A^T * B^T$

Grids			Mflops	
d	n	c	Bernoulli	Bik
2	10	1	1.412	0.121
2	17	1	1.372	0.042
2	25	1	1.321	0.019
3	10	1	1.001	XXXX
3	17	1	0.946	XXXX
3	25	1	0.918	XXXX

C is strong when its sparsity is precomputed, in order to narrow the number of iterations that need to be performed. This results in the performance difference for two reasons,

- Our compiler does not generate code to insert and delete elements from C. Bik's compiler generates code that *reinitializes* the storage of C and then creates entries in C as the computation is performed.

- Our compiler generates code that traverses C in order to determine what iterations of the original loop need to be performed. This is safe to do, since the compiler knows that C has been preallocated. Bik's compiler is not free to do this, and cannot use the predicate on the sparsity of C when performing guard encapsulation.

The bottom line is this: our compiler was able to generate code to enumerate the storage of C in a case where his was not. This resulted in our compiler generating a loop that directly enumerated the non-zeros of a column of C, while his compiler generated a loop that enumerated all of the indices for the column of C, zero and non-zero. Thus, the loops that our compiler generates have complexity $O(\kappa^2 n)$, where κ is the expected number of non-zeros in a column, while the loops generated by his compiler have complexity $O(\kappa n^2)$. Since κ is primarily a function of the order of approximation used to discretize the problem, it will stay constant as the problem size is scaled. That can be seen in the performance numbers, as our performance stays relatively constant, while his performance drops quadratically.

Furthermore, there is a bug in the code generates by Bik's compiler for the $C = A^T * B^T$ problem that causes $O(n^2)$ space to be allocated for C. "XXXX" is used to mark the entries in which this resulted in more memory being required than could be allocated. Bik has told us that this bug has been fixed in a more recent version of his compiler ([81]).

Chapter 18

Conclusions

We will close this thesis by, first, recounting its contributions to the body of knowledge. Then, we will discuss some problems and limitations of the work described here. Finally, we will suggest future areas of research.

18.1 Contributions

The concept of sparse compiling is relative new, having been first suggested by Bik in [24]. Because of this, there is a tremendous amount of this problem area that has been unexplored. Bik has presented one set of goals and design for approaching this problem, and we have presented another different set of goals and designs. Here are the key differences between our work and his, which constitute the contributions of this thesis,

Relational model. We have presented a set of analogies between aspects of spare compilation and relational databases, which we have called the relational model. This model has proven to be very useful for two basic reasons. First, having analogies with relational databases gives us an entire set of vocabulary with which to pose our problems and an entire set of literature from which to draw solutions. The second, is that the "sparse matrices as relations" analogy gave a us a powerful abstraction for sparse matrices, which was useful in the next point.

The black-box protocol. One of the major differences between our work and Bik's is that we allow the user to completely specify the storage format of sparse matrices, even going so far as allow the user to add new storage formats to the compiler. The black-box protocol was presented as the interface between the user's storage format implementation and the compiler. This protocol was shown to be more complicated to use than, say, source code annotations, but considerably more expressive.

An integrated inner join scheduler. In this thesis, we have shown that hierarchies of indices can be constructed from a storage format's access methods. We also developed a linear framework that can be used to discover join surfaces from a query's set of affine constraints. Then, we showed how both of these can be combined into a single non-deterministic algorithm for performing inner join scheduling. The advantage of such an algorithm is that it can easily be made deterministic by applying heuristics or other techniques. This is demonstrated in the current implementation of the compiler. We have shown that this basic approach can be extended to handle general queries, as well as fill and annihilation.

Complete Class I & II sparse compiler design. We have developed and presented a complete design for transforming dense specifications containing do-any loop nests into efficient sparse implementations. Since our design is being based upon a data-centric approach and, in particular, the concept of query optimization from the relational database literature, it can handle more general storage formats than Bik's iteration centric approach.

The Bernoulli sparse compiler. We have discussed our current implementation of this design, the Bernoulli sparse compiler. We have presented the heuristics that it uses and discussed its limitations. We have shown that it generates code for inner join queries that is competitive with real hand-written code. In particular, we have shown that it obtains performance levels comparable with PETSc and BlockSolve, two widely used sparse libraries. This, we claim, validates our basic design.

Enabling technology. By itself, our techniques for sparse compilation are of limited interest. But, perhaps most exciting about having this work in place is that it gives us a solid foundation upon which to build even more impressive edifices. The idea of viewing sparse computations as queries is very compelling: it allows programming systems to be built that express sparse codes using queries, an intentional representation of the computation, knowing that a sparse compiler is available to render them into efficient implementations.

18.2 Limitations of the current work

Of course, our design is not without its faults. In this section, we will present some of the warts on our work.

18.2.1 Limitations of the black-box protocol

The current version of the black-box protocol does not convey all of the information about storage formats that the compiler can use.

Ordering information. In order to schedule sort and merge joins–and in order to handle loops with dependencies, which we have not discussed–the compiler needs to know whether a given combination of access methods will enumerate the values of a field in a particular order. The current black-box protocol does not provide this information. Since it is not possible to detect when sorting is unnecessary, it is not possible to implement sort and merge joins efficiently.

Creation and deletion of non-zero entries. The current protocol does not provide for accessed methods to handle fill and annihilation. It would be relatively easy to extend the protocol to include access methods to

- pack and unpack the storage format from a canonical dynamic data structure, like a hash table, and
- insert and delete individual non-zero entries, if the format allowed for that.

Each of these are the extreme points of a spectrum. On one end is packing and unpacking matrices, which entails allocating and deallocating *all* of the non-entries of a sparse matrix at once. On the other end are the methods for allocating and deallocating a *single* entry of a sparse matrix. As Bik and Sparse MATLAB demonstrates with their operations for scattering and gathering entire rows or columns at a time, there are very important points between these two extremes. The real challenge in extending the protocol to handle fill and annihilation is not handling the two extremes, but in classifying and codifying all of the useful cases in between.

18.2.2 Limitations of our heuristic approach

The second major limitation of the current implementation is the way in which its non-deterministic portions have been made deterministic. In the current compiler, heuristics have been used to make "best guesses" at each decisions point. However, backtracking has not been incorporated into the implementation. Thus, if the compiler guesses "wrong" then it will can end up making a series of decisions that put it into an impossible situation. This happens particularly often during join scheduling: the compiler will make a sequence of decision and end up being deadlocked, even though there is a different sequence of decision that it could have made to avoid deadlock. This situation occurs quite frequently and can be very frustrating for the user, because the compiler does not give any sort of diagnostics indicating how the deadlock could have been prevented.

In addition to implementing some form of backtracking to prevent deadlock, our heuristics need to be further developed. We have already pointed out in the thesis known problems with our heuristics, and as the compiler is used more extensively, other will develop. Developing more and more robust heuristics will continue to be a constant challenge.

An alternative to relying on heuristics is to use search techniques from the field of artificial intelligence. However, the success of these methods depends upon having accurate cost models for comparing the performance of different sparse implementations. We have not developed such models, and we imagine that it will be difficult to do so. For instance, the performance of the final sparse implementation depends, to a large extent, upon the non-zero structure of the sparse matrices. Unfortunately, this is not known until runtime. Some sort of information about the non-zero structure would almost certainly have to be provided by the user in order to make an accurate estimate of performance. One way that this could be done is for the user to provide the compiler with several representative data sets that it can analyze.

18.2.3 Dependencies

One of our biggest limitations is the result of a design decision: we stated that we would only compile code containing doall loop nests, not loops with dependencies. There are several points in two primary points in our design at which dependencies have to be taken into account.

One point is during join scheduling. If a loop nest does not carry any dependencies, then P and Q can be chosen that will rearrange the iterations of the loop arbitrarily. If there are dependencies, then P and Q must chosen to satisfy the dependencies while still allowing an efficient high-level plan to be obtained.

Another point at which dependences come into point is during join implementation. If dependences require that the entries of a particular field be enumerated in a particular order, then either,

- The compiler must know that the entries are stored in that order and can be enumerated in that order. The ordering information that we have proposed adding to the black-box protocol can be used for this purpose.
- If the entries are not know to be ordered, then the compiler use generate code to sort them before enumerating them.

These issues are addressed by Kotlyar in [83]

18.2.4 Aggregation of structures

Another design decision was to only handle sparse matrices with a single underlying storage format. However, there are many interesting storage formats that are the composition of several storage formats. The BlockSolve format is an important one, but there are others, including the Yale format ([53]), in which the lower triangle of a matrix is CRS, the upper triangle in CCS, and the diagonal as a dense vector.

Bik has described a set of annotations in [25] that can be used for describing the sparsity for various regions of a sparse matrix. These annotations could almost certainly be merged with our annotations to include a description of the storage format for the region. However, the real challenge is not in describing the individual regions, but the interactions between them for a particular computation.

Consider the BlockSolve storage. This format first divides a sparse matrix into sets of columns which are labeled with a *color*. Then, within each color, the columns are further divided into *cliques*, where each clique had a dense block along the diagonal. Associated with each clique was a set of inodes that were not along the diagonal. This was illustrated in Figure 7.10. In addition to this structure, the BlockSolve library reordered the matrix to ensure that, within a color, none of the non-zeros from one inode fell in the same row as the non-zeros of another clique. When parallelizing the triangular solve computation, it is this property that allows the computations on all of the cliques within a single color to be done in parallel without any communication.

Bik's annotations can only be used to convey this property by describing the exact location of each clique and inode, and then having the compiler deduce the independence of the cliques within a color. This requires knowing the exact non-zero structure of the matrix at compile-time. A more useful set of annotations would allow the user to specify this as a property of the data structure and not of a particular sparsity structure.

18.2.5 Single vs. multiple field joins

We have repeatedly stated that, in order to schedule a query that describes a sparse computation for efficient evaluation, we want to scheduling it as a sequence of nested joins between single fields of relations. However, it is not clear that this will always yield the best implementation, nor is it clear that it is even safe to do so.

Considering the first point first, no, it is no clear that using a nested single field joins is always preferable. Consider the following code in which A and B are stored in CRS and CCS respectively.

```
for i := 1 to n do
    for j := 1 to n do
        sum := sum + A[i,j] * B[i,j];
    end do
end do
```

In this case, because A and B are stored with different orientations, it is not clear that scheduling joins on i and j separately is advantageous. The optimal plan for this code might be obtained by finding an implementation for the single join on both the i and j fields (e.g., put the entries of B into a hash table, enumerate the tuples of A and search the hash table).

At the moment, the current design and implementation will only attempt to schedule single field joins. We have taken this approach because understanding how to schedule single field joins is a prerequisite for scheduling multiple field joins. Also, we have not, at present, found a real (i.e., not contrived) instance in which using multiple fields joins might make a different. However, if such codes do exist, then it would not be too difficult to extend the techniques that we will discuss in this thesis to handle joins on multiple fields.

18.3 Future Work

Apart from addressing the limitations enumerated above, there are many other exciting directions of research. We discuss only a few here.

18.3.1 Optimization

In this thesis, we have discussed techniques for "scheduling" loop nests in order to achieve efficient sparse implementations. In places, we made reference to certain transformations that might be done to improve the results of our sparse compilation process. Now that our design for a sparse compiler is in place, we can begin to develop transformations and optimizations that will compliment these core techniques.

Some optimizations can be drawn from the database literature. One such optimization is to recognize expressions that occur more than one within a set of queries. Such expressions can be computed once, stored, and the stored result used instead of recomputing the expression ([120]). For instance,

```
for v ∈ A ⋈ B ⋈ C do
    ...
end do
for v ∈ A ⋈ B ⋈ D do
    ...
end do
```

might be transformed into,

```
T := A ⋈ B;
for v ∈ T ⋈ C do
    ...
```

```
end do
for v ∈ T ⋈ D do
    ...
end do
```

This transformation represents a space vs. time tradeoff, so there are undoubtedly some sophisticated criteria that must be developed in order to determine when it should be applied

Other optimizations can be taken from the compilers literature. One such optimization is reindexing. Suppose that the following low-level plan is produced by the join implementer.

```
while ... do
    ...
    for i := 1 to n do
        ... A[Search(v, i)] ...
    end do
    ...
end do
```

If it can be shown that the function call $Search(v, i)$ does not write to locations that are read in the while loop, then this loop can be rewritten as,

```
for i := 1 to n do
    w[i] := Search(v, i);
end do
while ... do
    ...
    for i := 1 to n do
        ... A[w[i]] ...
    end do
    ...
end do
```

If $Search(v, i)$ is expensive to evaluate, then this transformation can result in a significant increase in performance. In [84], we demonstrated that a sparse compiler could safely perform this optimization, and that it was worthwhile doing so. A similar optimization, called array slicing, is proposed for handling multiple levels of array indirection ([43]).

18.3.2 Packaging the technology

Another interesting question to be explored is, what are the ways in which to package this sparse compilation technology?

In this thesis, we have presented the Bernoulli sparse compiler. This compiler is like most other traditional batch compilers: it is invoked with a set of command line arguments, it reads a source file, it performs certain transformations to this code, and finally it writes some number of object files. However, this is not the only way that compiler technology can be embedded into compiling systems.

Another approach is to draw the user into the compiling process. That is, the compiling system can present the user with a GUI and allow the user to provide information and direction about particularly difficult sections of their code. Because of our very heavy reliance on heuristics in order to obtain efficient sparse implementations, and because there are many applications for which these heuristics might go wrong, the users of our compiler might be very grateful to have such a means of overriding decisions made by the compiler. ParaScope ([74, 63]) is a similar kind of system designed to allow the user control over the automatic parallelization of FORTRAN programs.

An even more radical approach is to automate the programming processes itself. There are several systems that allow the user to describe problems at the level of partial differential equations (PDE's). An automatic programming system is then used to apply the appropriate discretization techniques to reduce the PDE's to a system of linear equations. Then, an iterative solver is selected, based upon the discretization techniques used. Next, code for performing the computation is generated and executed. Finally, the results are presented to the user, usually in the form of a graphical display.

Such systems have been developed that target particular problem domains or particular types of PDE's. For instance, Parallel Ellpack ([107, 68, 67]) is designed to allow the user to solve elliptic PDE's without having to write any code. The PDE Toolbox in MATLAB ([94]) provides similar functionality.

A different approach to the problem of automatic programming is taken by SPL ([16, 123]). In this system, the user starts by expresses the PDE computation to be performed in an extremely high-level programming language. Then, the user directs the programming system through a sequence of transformations whose result is a program that can be run to produce the approximate solution. The real power of this system is its extensibility: if the system does not provide a transformation that the user requires, the user can provide the code to perform the transformation.

18.3.3 Parallelization

One of the most interesting uses of the sparse compilation technology is as a "code generator" for a parallelizing compiler.

One set of techniques based upon polyhedral algebra ([6]) has been developed for generating Single Program Multiple Data (SPMD) node programs from dense sequential source with HPF alignment and distribution directives ([56]). A different approach, the Inspector/Executor method ([122]), has been developed for generating SPMD node programs from sparse sequential source. What has been missing was a uniform framework for reasoning about dense *and* sparse SPMD code generation.

In [86], it is observed that many of the computations involved in scheduling communication and setting up message buffers in SPMD node programs can be expressed very easily as queries, if arrays, communication buffers, and distributions are all treated as abstract relations. By modeling these computations and data structures using the relational model, a unified framework for handling dense and sparse arrays, regular and irregular distributions, and so on, can easily be developed. Such an algorithm is presented in [86].

Another advantage of this approach is that it allows sparse matrix annotations to be added to HPF without altering the framework for generating code. Everything is still treated as a relation. The node programs produced by these techniques are nothing more than queries on relations that model sparse matrices. Our sparse compiler is a natural tool for transforming these queries into efficient, sparse SPMD node programs.

This approach to SPMD code generation will be elaborated upon in [82].

Part V

Appendices

Appendix A

The Matrices

The appendix describes the sparse matrices used that were used as data sets in this thesis.

A.1 Regular meshes

The matrices shown in Table A.1 were obtained by applying a first-order stencil to a regular d-dimensional grid, with n points in each dimension and c component variables at each grid point.

- Each row represents a different sparse matrix.
- The column labeled "N" gives the number of rows and columns in each matrix. The column labeled "Nzs" gives the number of non-zero entries the exist in each matrix.
- The "Goodness" column is the ratio of non-zeros entries to entries allocated by the Diagonal Skyline storage. This format, described in Section 7.3.3, is a cross between a dense and a sparse storage format and is designed to get the benefits of sparse storage without sacrificing dense performance. A high "Goodness" percentage represents a mesh that is amenable to dense storage, and hence dense performance. A low percentage represents a mesh that wastes a tremendous amount of space storing zero entries and, as a result, will not have good performance for many computations.

Table A.1: Regular Grids

d	n	c	N	Nzs	"Goodness" %
2	10	1	100	460	96.23
2	10	3	300	4140	69.49
2	10	5	500	11,500	65.83
2	10	7	700	22,540	64.37
2	17	1	289	1377	97.73
2	17	3	867	12,393	70.20
2	17	5	1445	34,425	66.45
2	17	7	2023	67,473	64.97
2	25	1	625	3025	98.44
2	25	3	1875	27,225	70.56
2	25	5	3125	75,625	66.78
2	25	7	4375	148,225	65.28
3	10	1	1000	6400	94.42
3	10	3	3000	57,600	64.25
3	10	5	5000	160,000	60.39
3	10	7	7000	313,600	58.88
3	17	1	4913	32,657	96.68
3	17	3	14,739	293,913	65.65
3	17	5	24,565	816,425	61.69
3	17	7	34,391	1,600,193	60.14
3	25	1	15,625	105,625	97.73
3	25	3	46,875	950,625	66.31
3	25	5	78,125	2,640,625	62.31

A.2 PETSc test matrices

Table A.2 shows the characteristics of matrices provided with PETSc ([61], [10]) for checking the integrating of that system.

Table A.2: PETSc test meshes

Name	N	Nzs	"Goodness" %
`tiny`	5	5	100.00
`small`	36	156	93.98
`medium`	181	2245	61.83
`arco1`	1501	26,131	58.47
`arco4`	27,007	543,103	65.79
`cfd.1.10`	15,360	496,000	61.56
`cfd.2.10`	122,880	4,134,400	62.53

A.3 Matrix Market matrices

Table A.3 shows the characteristics of matrices obtained from the Matrix Market [27].

Table A.3: Matrix Market meshes

Name	N	Nzs	"Goodness" %
662_bus	662	2474	2.41
685_bus	685	3249	4.05
1138_bus	1138	4054	2.96
add20	2395	17,319	5.73
add32	4960	23,884	12.20
bcsstm27	1224	56,126	53.09
bcsstk31	35,588	1,181,416	1.51
bcsstk32	44,609	2,014,701	0.96
e05r0000	236	5856	31.27
gr_30_30	900	7744	97.85
mahindas	1258	7682	7.32
memplus	17,758	126,150	1.04
nos4	100	594	35.48
nos5	468	5172	21.33
nos6	675	3255	98.25
nos7	729	4617	93.82
orani678	2529	90,158	9.09
sherman1	1000	3750	55.36

Appendix B

The Bernoulli Meta Form – BMF

The BMF language serves several purposes.

First, it is the source language of the compiler. Originally, it was not intended to be the language that the compiler's user would write. Instead, it was envisioned that there would be front-ends for conventional languages, like FORTRAN ([5]) and MATLAB ([94]), that would translate from these languages to BMF. While this approach is certainly possible, and even desirable, we have found that when used with a macro preprocessor system, such as M4 ([75]), BMF is a usable language for rendering matrix algorithms.

Second, BMF is used as the compiler's intermediate language. More precisely, the Abstract Syntax Tree (AST) of a parse BMF program is the core data structure within the compiler. This correspondence makes it possible to read and output the AST at any point in the compilation process. This greatly facilitates debugging.

In this appendix, we will describe BMF and its annotations in detail. In the first section, we will describe the core BMF language. In the second, we will describe the annotations that are recognized by the current compiler.

B.1 BMF - the language

B.1.1 Lexical structures

In the following, names that appear in italics, like *THIS*, are token names that appear in the grammar.

Identifiers. The following regular expression describes the sequences of characters that constitute an identifier in BMF,

$$\begin{aligned} \mathit{IDENT} &= \mathit{INITIAL}\ \mathit{SUBSEQUENT}^{+} \\ \mathit{INITIAL} &= \mathtt{a}|\ldots|\mathtt{z}|\mathtt{A}|\ldots|\mathtt{Z}|\mathtt{\_}|\mathtt{\%} \\ \mathit{SUBSEQUENT} &= \mathit{INITIAL}|\mathtt{0}|\ldots|\mathtt{9} \end{aligned}$$

BMF is case-sensitive, so `for` is recognized as a keyword, but `For` and `FOR` are recognized as two distinct identifiers.

Numbers. BMF recognizes the usual formats for integer (*INT*) and floating point (*FLOAT*) constants.

Strings. Strings (*STRING*) in BMF are delimited with double quotes (“`"`”). A backslash (“\”) can be used to escape characters, like “`"`”, within strings. Also, “`\t`”, “`\b`”, “`\r`”, and “`\n`” have the same effect as in C ([76]).

Whitespace and comments. BMF is a free-form language, like C, and white space is only used to determine the division between lexical tokens.

A sharp (“`#`”) marks the beginning of a comment in BMF. The comment ends when the line ends.

B.1.2 Declarations and definitions

file
  : (external-declaration | global-definition)*
external-declaration
  : “`extern`” procedure-declaration
global-definition

: variable-declaration
| procedure-definition

A file consists of a sequence of declarations and definitions. Procedures may be marked as “`extern`” when their definitions appear in another file. The exact linking semantics for variables, external and defined, is unspecified at the moment.

procedure-declaration
: procedure-header parameter-declarations
procedure-header
: “`procedure`” *IDENT* (“`(`” ident-list “`)`”)? parameter-declarations
: “`parambegin`” variable-declaration* “`paramend`”
ident-list
: *IDENT* (“`,`” *IDENT*)* procedure-definition
: procedure-declaration block-statement

A procedure declaration is a header followed by a set of parameter declarations. Parameter declarations may contain references to other parameters. This is useful when one parameter is an array, for instance, and another is the array's length. The following is an example of this use,

```
procedure dot_prod(n, x, y)
    parambegin
        var n : int;
        var x : array [0..n] of double;
        var y : array [0..n] of double;
    paramend
```

It is required, however, that parameters be declared before they are referenced.

Parameters are passed by value, unless they are declared with the “`"ref"`” annotation. See Section B.2 for details.

variable-declaration
    : “`var`” *IDENT* “`:`” arbitrary-type variable-init? annotation-list? “`;`”
variable-init
    : “`:=`” expression

Variable declarations may contain an optional initializations. Parameters, however, may not such an initialization. Annotations may optionally be placed on variable declarations.

B.1.3 Types

arbitrary-type
    : scalar-type
    | concrete-array-type
    | abstract-array-type
scalar-type
    : “`bool`”
    | “`real`”
    | “`int`”
    | “`void`”

There are two kinds of array types in BMF, concrete and abstract. Concrete arrays are those that actually exist during the program execution. These are like the arrays that are found in C or FORTRAN. The BMF compiler provides an implementation (the usual one) and optimizations for these array types. Abstract arrays are one for which the user provides the implementation. They are “abstract” because the compiler does not, a priori, know they implementation. Abstract array declarations must have annotation that describes their storage implementation.

concrete-array-type
    : “`array`” “`[`” concrete-array-bound
        (“`,`” concrete-array-bound)* “`]`” “`of`” scalar-type
    | “`array`” “`(`” concrete-array-bound
        (“`,`” concrete-array-bound)* “`)`” “`of`” scalar-type

concrete-array-bound : (“`0`” | “`1`”) “`:`” expression

Concrete arrays are like those in either C or FORTRAN, consecutive locations in memory where scalar types are stored. Unlike C or FORTRAN, which arbitrarily require arrays to be row or column major, and 0 or 1 based indexed, BMF allows the user to specify both. This allows the user to painlessly generate code to interface with either C or FORTRAN.

The first form of the concrete array type specifier uses square brackets (“`[`” and “`]`”) to surround the bounds. This indicates that the array is to stored in a row major fashion. The second form uses parenthesis (“`(`” and “`)`”), which indicates that the array is to be stored in a column major fashion. The row or column orientation of an array is considered part of its type, and arrays that differ only in their orientation do not have the same types.

Either 0 or 1 may be used as the lower bound in an array type, but if the array has multiple dimensions, all of the lower bounds must be 0 or 1; they may not be mixed within the same array.

```
# Concrete array with C-style indexing
var A : array[0:n-1,0:n-1] of real;
# Concrete array with FORTRAN-style indexing
var B : array(1:n,1:n) of real;
# Correct, but does not correspond to either C or FORTRAN
var C : array[1:n,1:n] of real;
# error
var D : array(0:n,1:n) of real;
```

abstract-array-type
: “`array`” “`int`” “`^`” *INT* “`of`” scalar-type

The abstract array type indicates only how many index fields an array has and the type of the values stored in the array. Other information, such as the bounds of the array, are extracted from the storage annotation that must accompany this type specification.

```
# Abstract 2d array stored in the "crs" format.
var A : array int^2 of real << << storage: "crs"
    n: (n) nzs: (anzs) rowptr: (arowptr)
    colind: (acolind) value: (avalue) >> >>;
# Concrete 1d array.
var y : array [0:n-1] of real;
# Abstract 1d array stored in the "sparsev" format.
var x : array int^1 of real << << storage: "sparsev"
    n: (n) nzs: (xnzs) index: (xind) value: (xval) >> >>;
```

B.1.4 Expressions

expression-list
: expression (“,” expression)*

expression
: “(” expression “)”
| expression “||” expression
| expression “&&” expression
| expression “==” expression
| expression “!=” expression
| expression “<=” expression
| expression “<” expression
| expression “>” expression
| expression “>=” expression
| expression “+” expression
| expression “-” expression
| expression “*” expression
| expression “/” expression
| “!” expression
| “-” expression
| “sqrt” “(” expression-list “)”
| relation-access
| set-expression
| quantified-expression
| “true” | “false” | *INT* | *FLOAT* | *STRING*
| *IDENT*

The operators and precedence rules in BMF are similar to those in C.

relation-access
: *IDENT* "`(`" expression-list "`)`"
| *IDENT* "`[`" expression-list "`]`"
| *IDENT* "`[`" "`[`" expression-list "`]`" "`]`"

Instead of having a single form of array access, BMF has several forms of more general relation access. The form that corresponds to conventional array access is the form with double square brackets ("`[[` and `]]`"). In this case, as in other languages, the element accessed is required to exist in the array, or a run-time error will occur. In this thesis, this form was referred to as *strong access*.

However, arrays in BMF are actually better thought of as relations with elements corresponding to tuples. Sparse arrays are relations in which not every element within the array's bounds exists. In this case, we need operators for testing for the existence of elements and for inserting new elements. The two other forms of relation access provide these. The parenthesis form ("`(`" and "`)`") returns a boolean indicating whether or not a particular element exists within a relation. This form is referred to as *lookup access*.

The remaining form, which uses single square brackets ("`[`" and "`]`"), is called *weak access*. It is like strong access, in that it can be used to refer to a particular element of a relation, but, unlike strong access, weak access does not require the particular element to exist within the relation. Here are the rules for evaluating weak access,

1. If the element exists within the relation, then it is referenced.
2. Otherwise, if the reference appears on the left-hand side (LHS) of an assignment statement, then the element is inserted into the relation and initialized with the value of the right-hand side (RHS).
3. Otherwise, if the reference appears in RHS position, a default value is generated, depending upon the base type of the relation,

```
int  → 0
real → 0.0
bool → false
```

4. The meaning of weak references in other positions, (e.g., as a parameter to a function), is unspecified at this time.

To illustrate the use of these various access forms, the following two codes, which implement the dot-product of two sparse vectors, will evaluate to the same result, although their execution may be quite different.

```
sum := 0;
for i := 1 to n {
    sum := sum + x[i] * y[i];
}
```

```
sum := 0;
for i := 1 to n {
    if (x(i) && y(i)) {
        sum := sum + x[[i]] *  x[[i]];
    }
}
```

The strong and lookup access forms are produced during compilation, and the user is not expected to use them directly. Instead, the evaluation rules for weak access are such that sparse algorithms can be written without having to worry about the management of elements in relations. Another way to say this is that, the user might write the first dot-product code given above, and the compiler might translate it into the second.

If a string constant appears as the first argument of a relation access, then this denotes the invokation of the access method within the relation whose name is given by the string argument.

```
A[["lookup_i", i, jj]]
```

This form is used by the compiler, and should not be used directly by the user (See Chapter 7 for its analog in the black-box protocol).

set-expression
    : "{" quant-expression-frame "|" expression "}"
quantified-expression
    : "(" quantifier quant-expression-frame "|" expression ")"
quantifier
    : "forall"
    | "exist"
quant-expression-frame
    : "[" (quant-ident ("," quant-ident)*)? "]"
quant-ident
    : *IDENT*
    | *IDENT* ":" scalar-type
    | "var" *IDENT* ":" scalar-type

BMF also provides syntax for quantified and unquantified sets of integers. The compiler is responsible for generating efficient evaluation schedules for these expressions, but the user is responsible for making the sets sensible. For instance, if the user does not give enough constraints to make a set finite, then the compiler may not produce sensible results. For instance,

```
do { [ i ] | true } {
    if ( 1 <= i && i <= 10 ) {
        ... } }
```

may not produce the same results as,

```
do { [ i ] | 1 <= i && i <= 10 } {
    ... }
```

The user should only use the first form of quantified identifier, which is just an unqualified *IDENT*. The later two forms are produced by the compiler, and express sets of locations and non-integer values.

B.1.5 Statements

block-statement
: "`{`" variable-declarations* statements* "`}`"
statement
: block-statement
| expression "`:=`" expression "`;`"
| "`while`" "`(`" expression "`)`" block-statement
| "`if`" "`(`" expression "`)`" block-statement else-clause?
| call-statement
| for-loop-statement
| do-loop-statement
| directive-statement
else-clause
: "`else`" conditional-statement
| "`else`" block-statement

The syntax of statements in BMF is similar to C, except that the sub-statement clauses of "`while`" and it must be surrounded by braces ("`{`" and "`}`").

call-statement
: (expression "`:=`")? "`call`" *IDENT* ("`(`" expression-list "`)`")? "`;`"

Function and procedure calls may only occur using the "`call`" syntax; they may not occur within expressions. This restriction has placed in order to simplify dependence analysis.

for-loop-statement
: "`for`" *IDENT* "`:=`" expression "`:`"
expression ("`:`" expression)? block-statement

The "`for`" syntax is similar to MATLAB's. The upper bound is inclusive, and the default step size is 1. The order of execution of the iterations is the usual one.

do-loop-statement
: "do" do-test block-statement
do-test
: set-expression
| "(" expression ")"

The "do" loop is used to enumerate the values within a set. The order of enumeration is unspecified, but the user may assume that the all side-effects from an iteration will be finalized before the next iteration is started. Basically, this syntax allows the user to express do-any loop nests.

directive-statement
: annotation-list statement

The directive syntax allows annotations to be placed on statements.

B.1.6 Annotations

annotation-list
: "<<" annotation-clause* ">>"
annotation-clause
: annotation-value
| ref-clause
| sparse-clause
| storage-clause
annotation-value
: "<<" annotation-value* ">>"
| block-statement
| "(" expression ")"
| "[" arbitrary-type "]"
| "STRING"
| *IDENT* ":"

The annotation syntax is an extremely general mechanism for expressing information that is otherwise outside of BMF's syntax. The last two forms of annotation value are equivalent. That is, "foo:" is equivalent to ""foo"".

B.2 BMF - the annotations

When the compiler examines an annotation list, it recognizes three specific annotation pairs and ignores the rest.

B.2.1 The “`"ref"`” annotation

ref-clause
: “`<<`” “`"ref"`” “`>>`”

A “`"ref"`” annotation on a parameter declaration indicates that the parameter is to be passed by reference. In the current implementation, copy-in/copy-out semantics are used instead of true call-by-reference semantics.

B.2.2 The “`"sparse"`” annotation

sparse-clause
: “`<<`” “`"sparse"`” “`>>`”

A “`"sparse"`” annotation on a parameter or variable array declaration indicates that the array is sparse. That is, not every element within the array's bounds is stored.

B.2.3 The “`"storage"`” annotation

The “`"storage"`” annotation is used to describe the storage implementation of abstract arrays.

storage-clause
: “`<<`” “`storage:`” *IDENT* storage-arg* “`>>`”

The *IDENT*, called the *storage name*, gives the name of the black-box that is to be used to implement the abstract array. The storage arguments that appear after the black-box are the arguments to the black-box.

storage-arg
: *IDENT* ":" annotation-value

Instead of being positional, the storage arguments are named with keywords. That is, instead of requiring that they appear in a particular order, the *IDENT* ":" keyword indicates which formal argument is to matched with the value. For instances, the following two storage annotations are equivalent.

```
var A1 : array int^2 of real
    << << storage: "foo" a: (1) b: {int} c: "zabba" >> >>;
var A2 : array int^2 of real
    << << storage: "foo" b: {int} c: "zabba" a: (1) >> >>;
```

Appendix C

The Black-Box Protocol

The black-box protocol is used to describe user-provided sparse matrix storage formats to the compiler in an abstract manner. This is done by describing each matrix stored in a format as if it were a database relation. In particular, the black-box protocol conveys the following information about each storage format,

- the schema (implicitly),
- the mapping between array references and the field of the schema,
- the bounds of the entries stored, and
- the access methods provided by the storage format.

A high-level explanation of the black-box protocol can be found in Chapter 7. In this appendix, we describe the protocol as it is currently implemented.

C.1 The Protocol

As was discussed in Chapter 7, the black-box protocol is actually the interface between storage format modules and the rest of the compiler. In this section, we present `bb.nw`, the source file from the Bernoulli compiler that contains this interface. The code is written in the Caml dialect of ML, a good introduction of which is [95] It was converted to LaTeX using the `noweb` literate programming tool ([106]).

C.1.1 Overview

The concrete array type is the only kind of array for which the compiler has a built-in storage implementation. All other storage implementations are provided to the compiler as modules that implement a defined interface, the black-box protocol.

At the beginning of the compiler's execution, each of the modules invokes the function `register` in order to enter their name and function for making black-boxes with the compiler.

⟨*BB protocol functions*⟩≡
```
value register : string -> black_box_maker -> unit;;
```

Then, when the compiler encounters a "`"storage"`" annotation, it looks up the storage name to find the corrisponding black-box. The function `is_registered` returns true or false, depending upon whether a particular name has been registered. The function `find_registered` returns the black-box that has been registered under a particular name. `find_registered` throws the exception `Not_found`, if no black-box has been registered under that name.

⟨*BB protocol functions*⟩+≡
```
value is_registered : string -> bool;;
value find_registered : string -> black_box_maker;;
```

A module registers a function of type `black_box_maker` with the compiler, and it is intended to construct black-boxes that describe the implementation for a particular abstract array.

⟨`black_box_maker` *type*⟩≡
```
type black_box_maker == keyword_map -> black_box
and keyword_map == (string, annotation) map__t
;;
```

Here are the steps that occurs when the compiler processes a “`"storage"`” annotation,

1. The annotation is checked for syntactic correctness, and the black-box name and list of keywords and values, hereafter called the keyword arguments, are collected.

2. The compiler searches for a black-box maker that is registered under the given black-box name. If not such maker is found, then the compiler signals an error.

3. The keyword arguments are inserted into an associate map, which has type `keyword_map`. The black-box maker is called with this map as its sole argument.

4. The black-box maker, using the keyword map, constructs a black-box and returns it to the compiler.

C.1.2 The Black-Box

The black-box objects returned by the black-box makers have type `black_box`.

⟨`black_box` *type*⟩≡
```
type black_box = {
     bb_index_fields : string list;
     bb_value_field : string;
     bb_bounds_maker : bounds_method;
     bb_access_methods : access_method list;
}
and bounds_method == bmf_expr list -> bmf_expr
;;
```

A black-box object contains all of the information about a particular abstract array that the compiler needs to schedule, optimize, and generate references to the array. The abstraction that is presented to the compiler is that the abstract array is a relation, with

- a set of fields, and
- a set of of methods for accessing relation's tuples (the *access methods*).

The fields of a `black_box` record contain the following information,

- `bb_index_fields` and `bb_value_field`: These two values constitute the mapping, as described in Section 7.1, between the positions of an array reference and the fields of the relation representing the array. `bb_index_fields` are the fields that match the array reference indices that appear in the user's source code, and `bb_value_field` is the field that contains values of the reference. That is, if array `A` has index fields `A.i` and `A.j`, and value field `A.v`, then the array reference `A[x,y]` will match the tuple in `A` whose `A.i` field is `x` and whose `A.j` field is `y`, and will refer to the location of the `A.v` field within that tuple.

- `bb_bounds_maker`: This is a function for generating an expression that describes the bounds on the index fields of the relation. More specifically, this field contains a function that takes a list of expressions, one for each index field, and returns a set expression that describes the contraints on the index fields in terms of the parameter expressions. For instance, if `A` is a square `n`×`n` matrix, and if its bounds maker is called with the expressions `i` and `j`, then it might return the BMF expression,

  ```
  { [i,j] | 0 <= i && i <= n-1 && 0 <= j && j <= n-1  }
  ```

  Note that a bounds maker does not have to return rectangular bounds. If `B` was only defined for the lower triangle of `A`'s bounds, and its bounds maker is called with the expressions `p` and `q`, it might return the BMF expression,

  ```
  { [p,q] | 0 <= p && p <= n-1 && 0 <= q && q <= n-1
                 && p >= q }
  ```

There is nothing within the protocol that ensures that these contrains are even linear. However, in practice, they must be.

- `bb_access_methods`: A list of the methods that are available for accessing the tuples of the relation, which are described below

C.1.3 Access Methods

The access methods are the mechanisms by which tuples of a relation are accessed.

⟨access_method *type*⟩≡

```
type access_method = {
    am_name : string;
    am_in_fields : string list;
    am_out_field : string;
    am_info : access_method_kind
}

and access_method_kind =
    am_fun of am_fun_type
|   am_rel of am_rel_type
;;
```

The `am_name` field is the name of the access method. This name, which has no connection with either the array's name or the names of the array's fields, is simply a tag for refering to a particular access method. When a reference to a access method appears in a BMF program, the access method's name appears as the reference's first argument. For instance,

```
A[["lookup_i", i, jj]]
```

refers to the result of invoking the access method within `A`'s black-box whose `am_name` field is `"lookup_i"`, with the arguments `i` and `jj`.

When an access method is invoked, the method returns all of the values of the `am_out_field` of all of the tuples in which the `am_in_field` match the arguments to the access method. This can be expressed more clearly using the relational algebra: the reference `A[["lookup_i",i,jj]]`, where the `"lookup_i"` method of `A` has infields `A1` and `A2`, and output field `A3`, is equivalent to

$$\pi_{\mathtt{A3}}\sigma_{\mathtt{A1}=\mathtt{i},\mathtt{A2}=\mathtt{jj}}\mathtt{A}$$

The `am_info` field indicates the cardinality of the result of the access method. In general, an access method will return a stream of all of the tuples that match the values of the input fields. This is indicated by an `am_rel` value in the `am_info` field. But sometimes it is the case that the values of the input fields uniquely determine the value of the output field. That is, only one tuple will be found in the relation for the any combination of values of the infields. In this case, for a set of values of the input fields, the access method could only return a single value for the output field. this single result is refered to as a singleton, and this type of constraint on the values of the fields of the relation is refered to as a *functional dependency* in the database literature. It indicated by a `am_fun` value in the `am_info`.

It is important to realized that, although a functional dependency implies that there cannot be more than one tuple for a particular set of values of the input fields, it does *not* imply that a tuple with those values exists in the relation. It simply implies that, if there do any tuples with those values, then there is only one.

C.1.4 Functional Access Methods

A functional access method is used when there is a singleton result that can match the given values of the input fields. In this case, the compiler needs to know the cost of accessing the tuple, and how to generate code for accessing the tuple. The `am_fun_cost` of a functional access method is the expected cost of execution the code associated with the access method. There are two ways of specifying how the code for the method can be generated.

```
⟨1am_fun_type type⟩≡
type am_fun_type ==
     am_fun_cost    (* cost of access *)
   * am_fun_kind    (* func for generating code *)

and am_fun_cost = O_1 | O_logn | O_n

and am_fun_kind =
     am_fun_search of am_search_type
|    am_fun_lookup of am_lookup_type
;;
```

The first interface is used to describe access methods that can either succeed or fail in finding an entry for a particular set of infield values. An example of this sort of access method is one that searches for a particular index within the entries stored in the storage format. If the indices stored are space, then the search may or may not succeed. A discussion of how this interface is used can be found in Section 13.3.

```
⟨2am_search_type⟩≡
type am_search_type ==
     bmf_expr list               (* args *)
     -> ( bmf_expr -> bmf_stmt ) (* found_f *)
     -> ( unit -> bmf_stmt )     (* not_found_f *)
     -> bmf_stmt
;;
```

The second interface is used to describe access methods that can only succeed in finding an entry for a particular set of infield values. An example of this sort of access method is one that performs an array dereference. If the infields' values were correctly obtained, then the array dereference should always return a result. A discussion of how this interface is used can be found in Section 14.3.1.

```
⟨5am_lookup_type⟩≡
type am_lookup_type ==
    bmf_expr list              (* args *)
    -> ( bmf_expr -> bmf_stmt)  (* found_f *)
    -> bmf_stmt
;;
```

C.1.5 Relational Access Methods

A relational access method is used when an arbitrary number of tuples can match the given values of the input fields. In this case, the compiler needs code for enumerating these tuples. Unlike an `am_fun_type`, an `am_rel_type` does not have any cost information: the current protocol assumes, perhaps incorrectly, that the cost of enumerating a set of tuples is proportional to the number of tuples. In practice, this appears to be a reasonable assumption.

```
⟨7am_rel_type type⟩≡
type am_rel_type =
    am_rel_interval of am_range_type
|   am_rel_general of am_stream_type
;;
```

There are two ways of specifying how code can be generated for relational access methods. The first, the `am_range_type`, is used when the values of the output field are exactly all integer values in closed interval, [`lb`...`ub`]. A discussion of how this interface is used can be found in Section 14.3.1.

```
⟨9am_range_type⟩≡
type am_range_type ==
    bmf_expr list
    -> ( bmf_expr(*lb*) -> bmf_expr(*ub*) -> bmf_stmt )
    -> bmf_stmt
;;
```

The second mechanism, the `am_stream_type`, allows a more flexible means of enumeration. A discussion of how this interface is used can be found in Section 13.2.

```
⟨am_stream_type⟩≡
type am_stream_type ==
    bmf_expr list
    -> am_handler_record

and am_handler_record = {
    decl_ids : bmf_ident list;
    init_st : bmf_stmt;
    incr_st : bmf_stmt;
    close_st : bmf_stmt;
    valid_ex : bmf_expr;
    deref_ex : bmf_expr }
;;
```

C.1.6 The source file `bb.mli`

Finally, here is how the code "chunks" given above fit together to form the source file `bb.mli`.

```
⟨bb.mli⟩≡
#open "bmf";;

⟨am_stream_type⟩
⟨am_range_type⟩
⟨am_rel_type type⟩
⟨am_search_type⟩
⟨am_lookup_type⟩
⟨am_fun_type type⟩
⟨access_method type⟩
⟨black_box type⟩
⟨black_box_maker type⟩

⟨BB protocol functions⟩
```

C.2 An extended example of the BB protocol

To illustrate the use of this protocol, we present the complete implementation of a black-box, namely the module that describes the implementation of sparse vectors.

A sparse vector is a relation with three fields, ii, i, and v, which hold the offset, dense index, and value of each non-zero entry, respectively. The core of the sparse vector black-box is the function, `sparsev_maker`, which generates black-box objects for sparse vector variables. When invokes with a set of black-box arguments, this function must,

- Parse these argument and extract the relevant information,
- Create closures for each of the access methods, and
- Create a record of all of the information that constitutes the black-box object.

⟨`sparsev_maker` *definition*⟩≡

```
let sparsev_maker args =
     let ⟨parse and bind arguments⟩
     in let ⟨sparse vector methods⟩
     in
          ⟨generate black-box object⟩
;;
```

Processing the arguments. There are four arguments that must be provided in order to assemble the black-box.

`n` – The upper bound of index field, i, which will range from 0 to $\mathtt{n} - 1$.

`nzs` – The number of entries stored in the sparse vector. The offset field, ii, will range in value from 0 to $\mathtt{nzs} - 1$.

`index` – The name of the dense integer vector which holds the dense index of each non-zero entry. The value of the i field associated with an offset, ii, is obtained by $\mathtt{index}[ii]$

`value` – The name of the dense real vector which holds the value of each non-zero entry. The value of the v field associated with an offset, ii, is obtained by $\mathtt{value}[ii]$

⟨*parse and bind arguments*⟩≡

```
n = required an_expr_arg args "n"
and nzs = required an_expr_arg args "nzs"
and index_ptr = required an_expr_var_arg args "index"
and value_ptr = required an_expr_var_arg args "value"
```

The access methods. With these four arguments, it is possible to create closures that will generate code for each of the four access methods that are provided for a sparse vector.

The access method, `enum_ii`, will produce a stream of the values of ii. More precisely, it produces the lower and upper bounds, 0 and $\mathtt{nzs}-1$, on the range of offsets. These bounds are passed to `body_f`, the function provided by the compiler for generate the code that uses these bounds.

The argument, `prereq`, will be the list of arguments that appeared in the access method invokation. The `match_list_0 prereq` expression is used to destruct this list. In this case, the invokation of this access method takes no arguments, so all that `match_list_0` does is check that `prereq` is a length zero list. However, in subsequent methods, `match_list_N` will return the `N` arguments from the `prereq` list.

⟨*sparse vector methods*⟩≡

```
enum_ii_m prereq body_f =
    let _ = match_list_0 prereq
    in
        body_f (expr_int(0)) (expr_op(op_plus,[nzs;expr_int(-1)]))
```

The access methods, `lookup_i` and `lookup_v`, are used to dereference an *ii* offset to obtain the associated *i* and *v* values. `body_f` is the function provided by the compiler to generate the code that uses the results of the dereference.

⟨*sparse vector methods*⟩+≡

```
and lookup_i_m prereq body_f =
    let ii = match_list_1 prereq
    in
        body_f (parse_expr "$indp[[$ii]]"
                    [("$indp",an_string(index_ptr));
                     ("$ii",an_expr(ii))])
and lookup_v_m prereq body_f =
    let ii = match_list_1 prereq
    in
        body_f (parse_expr "$valp[[$ii]]"
                    [("$valp",an_string(value_ptr));
                     ("$ii",an_expr(ii))])
```

The access method, `search_i`, uses a binary search to find the *ii* offset associated with a particular value of *i*. The compiler supplied function, `found_f`, is invoked to generate the code for when such an *ii* is found, and the function, `not_found_f`, is invoked to generate the code for when case when it is not found.

⟨*sparse vector methods*⟩+≡

```
and search_i_m prereq found_f not_found_f =
    let i = match_list_1 prereq
    in
        gen_binary_search i (expr_int(0)) nzs index_ptr
            found_f not_found_f
```

The black-box object. With the closures for the access methods in place, the black-box object that will be returned by `sparsev_maker` can be constructed. It contains the schema of the relation, the arrays bounds of the vector, in terms of the dense index field, i, and the list of available access methods.

⟨1*generate black-box object*⟩≡

```
{
     bb_index_fields = ["i"];
     bb_value_field = "v";
     bb_bounds_maker = (interval_bounds0 "n" args);
     bb_access_methods =
          (⟨generate the list of access methods⟩);
}
;;
```

The list of available access methods includes, for each method, the name, input fields, output fields, and code generation closure. For the singleton methods, an estimate of the cost of invoking the method is included as well.

⟨3*generate the list of access methods*⟩≡

```
[
     { am_name="enum_ii"; am_in_fields=[]; am_out_field="ii";
          am_info=am_rel(am_rel_interval(enum_ii_m)) };

     { am_name="lookup_i"; am_in_fields=["ii"]; am_out_field="i";
          am_info=am_fun(O_1,am_fun_lookup(lookup_i_m)) };

     { am_name="lookup_v"; am_in_fields=["ii"]; am_out_field="v";
          am_info=am_fun(O_1,am_fun_lookup(lookup_v_m)) };

     { am_name="search_i";am_in_fields=["i"]; am_out_field="ii";
          am_info=am_fun(O_logn,am_fun_search(search_i_m)) }
]
```

The source file. This implementation of the sparse vector storage format is found in the source file `sparsev.ml`. In addition to the definition of `sparsev_maker`, this file contains a call to `bb__register` in order to register this black-box module with the sparse compiler.

```
⟨sparsev.ml⟩≡
#open "util";;
#open "bmf";;
#open "bmf_parser_aux";;
#open "bb";;
#open "bb_utils";;

⟨sparsev_maker definition⟩

register "sparsev" sparsev_maker;;
```

C.3 Future work

There are several features that are missing from the current black-box implementation that would be required in order to implement some of the material discussed in this thesis.

- The combining operator. This is the operator that is used in order to combine multiple entries with the same array index to form the single value associated with the array index. Since this is not currently specified, it is currently assumed to be "no duplicates", which means that multiples entries with the same array index cannot be stored within a relation. This is discussed in Section 7.1.4.

- An indication of whether or not values of join fields produced by combinations of methods, or more precisely indexing enum terms, are produced in sorted order. The information is necessary in order to eliminate the "sort" phase of sort and merge joins. This is discussed in Section 12.5.1.

- Methods for packing, or creating an instance of a storage format by the entries in a hash table or other canonical data structure. These methods are required in order to generate code that handles fill and annihilation. This is discussed in Section 16.2.

BIBLIOGRAPHY

[1] R.C. Agarwal, F.G. Gustavson, and M. Zubair. A high performance algorithm using pre-processing for the sparse matrix-vector multiplication. In *Supercomputing '92*, pages 32–41, Minneapolis, Minnesota, November 16–20, 1992.

[2] E. Allman et al. Embedding a data manipulation language in a general purpose programming language. In *Proceedings of the 1976 ACM-SIGPLAN-SIGMOD Conference on Data Abstraction, Definition and Structure*, Salt Lake, Utah, March 1976.

[3] Fernando L. Alvarado, Alex Pothen, and Robert Schreiber. Highly parallel sparse triangular solution. In J.A. George, J.R. Gilbert, and J.W.H. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms*, IMA Volumes in Mathematics and its Applications #58, pages 141–158. Springer-Verlag, Berlin, 1993.

[4] American National Standards Institute. *ANSI X3.198-1992 – Programming Language – Fortran – Extended.*

[5] American National Standards Institute. *ANSI X3.9-1978 – Programming Language FORTRAN.*

[6] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A linear algebra framework for static HPF code distribution. In *Proceedings of the 4th Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993. This paper has subsequently been revised.

[7] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide - Release 2.0.* Society for Industrial and Applied Mathematics, 1994.

[8] E. C. Anderson and Y. Saad. Solving sparse triangular systems on parallel computers. *International Journal of High Speed Computing*, 1:73–96, 1989.

[9] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, PA, May 21–24, 1996.

[10] Satish Balay, William Groopp, Lois Curfman McInnes, and Barry Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 – Revision 2.0.15, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[11] Utpal Banerjee. *Depedence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, MA., 1988.

[12] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Norwell, MA., 1993.

[13] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: Loop Parallelization.* Kluwer Academic Publishers, Norwell, MA., 1994.

[14] Randolph E. Bank and Craig C. Douglas. Sparse matrix multiplication package (SMMP). *Advances in Computational Mathematics*, 1:127–137, 1993.

[15] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1993.

[16] Gal Berkooz, Paul Chew, Jim Cremer, Rick Palmer, and Richard Zippel. Generating spectral method solvers for partial differential equations. Tr 92-1308, Department of Computer Science, Cornell University, October 1992.

[17] Gautam Bhargava, Piyush Goel, and Bala Iyer. Hypergraph based reordering of outer join queries with complex predicates. In *ACM SIGMOD International Conference on Management of Data*, pages 304–315, San Jose, CA, May 23–25, 1995. in SIGMOD Record 24(2), June 1995.

[18] Gautam Bhargava, Piyush Goel, and Bala Iyer. Efficient processing of outer joins and aggregate functions. In *Proceedings of the 12th International Conference on Data Engineering*, pages 441–449, New Orleans, LA, February 26 – March 1, 1996.

[19] Aart J. C. Bik. *Compiler Support for Sparse Matrix Computations.* PhD thesis, University of Leiden, Leiden, The Netherlands, May 1996.

[20] Aart J.C. Bik, Peter M.W. Knijnenburg, and Harry A.G. Wijshoff. Reshaping access patterns for generating sparse codes. In *The Seventh International Workshop on Languages and Compilers for Parallel Computing*, pages 406–422, Ithaca, NY, August 8–10, 1994.

[21] Aart J.C. Bik and Harry A.G. Wijshoff. Advanced compiler optimizations for sparse computations. In *Supercomputing '93*, pages 430–441, Portland, OR, November 15–19, 1993.

[22] Aart J.C. Bik and Harry A.G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the 1993 International Conference on Supercomputing*, pages 416–424, Tokyo, Japan, July 20–22, 1993.

[23] Aart J.C. Bik and Harry A.G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 12–14, 1993.

[24] Aart J.C. Bik and Harry A.G. Wijshoff. A sparse compiler. Technical Report 93–04, Dept. of Computer Science, Leiden University, 1993.

[25] Aart J.C. Bik and Harry A.G. Wijshoff. Annotations for a sparse compiler. In *The Eight International Workshop on Languages and Compilers for Parallel Computing*, pages 500–514, Columbus, OH, August 10–12, 1995.

[26] Aart J.C. Bik and Harry A.G. Wijshoff. Simple quantitative experiments with a sparse compiler. In *Parallel algorithms for irregularly structured problems: third international workshop, IRREGULAR '96*, pages 249–262, Santa Barbara, CA, August 19–21 1996.

[27] R.F. Boisvert, R. Pozo, K. Remington, R.F. Barrett, and J.J. Dongarra. *The Quality of Numerical Software: Assessment and Enhancement*, chapter Matrix Market: a web resource for test matrix collections, pages 125–137. Chapman and Hall, London, 1997.

[28] William L. Briggs. *A Multigrid Tutorial*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987.

[29] Peter Brinkhaus, Aart J.C. Bik, and Harry A.G. Wijshoff. Subroutine on demand-service, sparse blas 2 & 3. `http://hp137a.wi.leidenuniv.nl:8080/blas-service/blas.html`. Accessed January 28, 1997.

[30] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175, Palo Alto, CA, June 25–27 1986. appeared in SIGPLAN Notices 21(7), July 1986.

[31] Sandra Carney, Michael A. Heroux, Guangye Li, and Kesheng Wu. A revised proposal for a sparse BLAS toolkit. SPARKER Working Note #3, June 1994. `http://www.cerfacs.fr/~douglas/mgnet/papers/SparKer/sparker3.ps.gz`.

[32] D. D. Chamberlin et al. SEQUEL 2: a unified approach to definition, manipulation, and control. *IBM Journal of Research and Development*, 20(6):560–575, 1976.

[33] D. D. Chamberlin et al. A history and evaluation of System R. *Communications of the ACM*, 24(10):632–646, 1981.

[34] Donald D. Chamberlin, Morton M. Astrahan, W. Frank King, Raymond A. Lorie, J. E. Mehl, Thomas G. Price, Mario Schkolnick, Patricia G. Selinger, Donald R. Slutz, Bradford W. Wade, and Robert A. Yost. Support for repetitive transactions and ad-hoc queries in System R. *ACM Transactions on Database Systems*, 6(1):70–94, 1981.

[35] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *OOPSLA '89 Conference Proceedings*, 1989.

[36] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[37] E.F. Codd. Extending the database relational model to capture more meanings. *ACM Transactions on Database Systems*, 4(4):387–434, December 1979.

[38] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer-Verlag, 1993.

[39] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 57–66, Atlanta, Georgia, June 22–24, 1988.

[40] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[41] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.

[42] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattic model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 17–19, 1977.

[43] Raja Das, Joel Saltz, and Reinhard van Hanxleden. Slicing analysis and indirect accesses to distributed arrays. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, pages 152–168, Portland, Oregon, August 12–14, 1993. Also as University of Maryland Technical Report CS-TR-3076 and UMIACS-TR-93-42.

[44] C.J. Date. The outer join. In *Proceedings of the Second International Conference on Databases*, pages 76–106, Cambridge, England, August 30–September 3, 1983.

[45] Michael M. David. Advanced capabilities of the outer join. *ACM SIGMOD Record*, 21(1):65–70, 1992.

[46] Jack Dongarra, Andrew Lumsdaine, Xinhiu Niu, Roldan Pozo, and Karin Remington. A sparse matrix library in C++ for high performance architectures. In *Proceedings of the Second Object Oriented Numerics Conference*, pages 214–218, 1994.

[47] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[48] J.J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):18–32, March 1988.

[49] C.C. Douglas and W.L. Miranker. Constructive interference in parallel algorithms. *SIAM Journal of Numerical Analysis*, 25:376–398, 1988.

[50] C.C. Douglas and B.F. Smith. Using symmetries and antisymmetries to analyze a parallel multigrid algorithm: The elliptic boundary value case. *SIAM Journal of Numerical Analysis*, 26:1439–1461, 1989.

[51] Iain S. Duff, Michele Marrone, Giuseppe Radicati, and Carlo Vittoli. A set of level 3 basic linear algebra subprograms for sparse matrices. Technical Report RAL-TR-95-049, Computing and Information Systems Department, Atlas Centre, Rutherford Appleton Laboratory, Oxon OX11 0QX, England, September 11, 1995.

[52] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, England, 1986.

[53] S.C. Eisenstat, H.C. Elman, M.H. Schultz, and A.H. Sherman. The (new) Yale sparse matrix package. In G. Birkoff and A. Schienstadt, editors, *Elliptic Problems Solvers II*, pages 45–52. Academic Press, New York, NY, 1984.

[54] Paul Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988.

[55] Marc Feeley and Guy Lapalme. Using closures for code generation. *Journal of Computer Languages*, 12(1):47–66, 1987.

[56] High Performance Fortran Forum. High performance fortran language specification. *Scientific Programming*, 2(1–2):1–170, 1993.

[57] Cong Fu and Tao Yang. Run-time compilation for parallel sparse matrix computations. In *Proceedings of the 1996 International Conference on Supercomputing*, pages 237–244, Philadelphia, PA, May 25–28, 1996.

[58] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. Technical Report CSL-91-4, Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, June 1991.

[59] John R. Gilbert and Robert Schreiber. Highly parallel sparse cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 13(5):1151–1172, 1992.

[60] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[61] William Gropp and Barry Smith. Scalable, extensible, and portable numerical libraries. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 87–93. IEEE, 1994.

[62] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce Lindsay, Hamid Pirahesh, Michael J. Carey, and Eugene Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–161, March 1990.

[63] Mary W. Hall, Timothy J. Harvey, Ken Kennedy, Nathaniel McIntosh, Kathryn S. McKinley, Jeffrey D. Oldham, Michael H. Paleczny, and Gerald Roth. Experiences using the ParaScope editor: an interactive parallel programming tool. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 33–43, San Diego, California, May 19–22, 1993.

[64] Michael T. Heath, Esmond Ng, and Barry W. Peyton. Parallel algorithms for sparse linear systems. In K.A. Gallivan et al., editors,

Parallel Algorithms for Matrix Computations, pages 83–124. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1990.

[65] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1990.

[66] M.R. Hestenes and E.L. Stiefel. Methods of conjugate gradient for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[67] E.N. Houstis, J.R. Rice, N.P. Chrisochoides, H.C Karathanasis, P.N. Papachiou, M.K. Samartizis, E.A. Vavalis, Ko Yamg Wang, and S. Weerawarana. //ELLPACK: A numerical simulation programming environment for parallel MIMD machines. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 96–107, Amsterdam, The Netherlands, June 11–15, 1990.

[68] E.N. Houstis, J.R. Rice, and T.S. Papatheodorou. Parallel ellpack: An expert system for parallel programming of partial differential equations. *Mathematics and Computers in Simulation*, 31:497–507, 1989.

[69] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, September 1984.

[70] Yannis E. Ioannidis. Query optimization. In Allen B. Tucker, Jr., editor, *CRC Computer Science and Engineering Handbook*, pages 1038–1057. CRC Press, LLC, 1997.

[71] Matthias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[72] Mark T. Jones and Paul E. Plassmann. The efficient parallel iterative solution of large sparse linear systems. In A. George, J. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56 of *IMA Volumes in Mathematics and Its Applications*, pages 229–245. Springer-Verlag, 1993. Also MSC Preprint P314-0692.

[73] Mark T. Jones and Paul E. Plassmann. Blocksolve95 users manual: Scalable library software for the parallel solution of sparse linear systems. Technical Report ANL-95/48, Argonne National Laboratory, December 1995.

[74] Ken Kennedy, Kathryn S. McKinley, and Chau-Wen Tseng. Analysis and transformation in the parascope editor. In *Proceedings of the 1991 International Conference on Supercomputing*, Cologne, Germany, June 17-21, 1991.

[75] B. W. Kernighan and D. M. Ritchie. The m4 macro processor. In *Supplementary Documents*, volume 2 of *Unix Programmer's Manual*. Bell Telephone Laboratories, Incorporated, Murray Hill, New Jersey, seventh edition, January 1979.

[76] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 1988.

[77] D. Kincaid, J. Respess, D. Young, and R Grimes. Algorithm 586 ITPACK 2C: A FORTRAN package for solving large sparse linear systems by adaptive accelerated iterative methods. *ACM Transactions on Mathematical Software*, 8(3):302–322, September 1982.

[78] Peter M.W. Knijnenburg and Harry A.G. Wijshoff. On improving data locality in sparse matrix computations. Technical Report 94–15, Leiden University, 1994.

[79] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computing Programming*. Addison-Wesley, Reading, MA, 1973.

[80] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 16–18, 1997.

[81] Vladimir Kotlyar. personal communication, 1996.

[82] Vladimir Kotlyar. *A Relational Approach to the SPMD Code Generation*. PhD thesis, Cornell University, December 1997. to appear.

[83] Vladimir Kotlyar and Keshav Pingali. Sparse code generation for imperfectly nested loops with dependences. In *Proceedings of the 1997 International Conference on Supercomputing*, Vienna, Austria, July 7–11, 1997. to appear.

[84] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Automatic parallelization of the conjugate gradient algorithm. In *The Eight International Workshop on Languages and Compilers for Parallel Computing*, LNCS #1033, Springer-Verlag, Columbus, OH, August 10–12, 1995.

[85] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. In *EUROPAR*, 1997.

[86] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Unified framework for sparse and dense spmd code generation (preliminary report). Technical Report TR97-1625, Department of Computer Science, Cornell University, March 1997.

[87] V. Kumar, A. Grama, A. Gupta, and G. Kapyris. *Parallel Computing*. Benjamin Cummings, Redwood, California, 1994.

[88] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45:255–282, 1950.

[89] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[90] Byung Suk Lee and Gio Wiederhold. Outer joins and filters for instantiating objects from relational databases through views. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):108–119, February 1994.

[91] P. E. Lewis and J. P. Ward. *The Finite Element Method: Principles and Applications*. Addison-Wesley, Reading, Massachusetts, 1991.

[92] Wei Li and Keshav Pingali. Access normalization: Loop restructuring for numa compilers. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*,

pages 285–295, Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, October 12–15, 1992.

[93] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, April 1994.

[94] The MathWorks Inc. *MATLAB Reference Guide*, 1992.

[95] Michel Mauny. *Functional Programming using Caml Light*, January 1995. Available from `http://pauillac.inria.fr/caml/`.

[96] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 2–15, Charleston, SC, January 10-13, 1993.

[97] Pacific-Sierra Research Corporation. VAST. `http://www.psrv.com/vast/`. Accessed June 22, 1997.

[98] Stacy Pendell. Sp configuration. `http://www.tc.cornell.edu/UserDoc/SP/config.html`, September 9, 1996. Accessed June 22, 1997.

[99] Alex Pothen and Chunguang Sun. Compact clique tree data structures in sparse matrix factorizations. In Thomas F. Coleman and Yuying Li, editors, *Proceedings of the Workshop on Large-Scale Numerical Optimization*, pages 180–204, Ithaca, NY, October 19–20, 1989. Published by the Society for Industrial and Applied Mathematics, Philadelphia, PA, 1990.

[100] Roldan Pozo. *MV++ v. 1.5a Matrix/Vector Class Reference Guide*, November 8, 1995.

[101] Roldan Pozo, Karin A. Remington, and Andrew Lumsdaine. *SparseLib++ v. 1.5, Sparse Matrix Class Library*, November 1995.

[102] Bill Pugh and Tatiana Shpeisman. Compiling efficient sparse matrix code: LU factorization. white paper, November 18, 1996.

[103] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[104] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, LNCS #768, pages 546–566, Portland, Oregon, August 12–14, 1993. Springer-Verlag.

[105] Raghu Ramakrishnan. *Database Management Systems*. McGraw-Hill, New York, NY, beta edition, 1996.

[106] Norman Ramsey. Literate programming simplified. *IEEE Software*, pages 97–105, September 1994.

[107] John R. Rice and Ronald F. Boisvert. *Solving Elliptic Problems Using ELLPACK*. Springer-Verlag, New York, NY, 1985.

[108] Youcef Saad. Krylov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, November 1989.

[109] Youcef Saad. *SPARSKIT: a basic tool kit for sparse matrix computations, Version 2*, June 6 1994.

[110] Youcef Saad and Martin Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[111] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, Boston, MA, 1996.

[112] P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access path selection in a relational database management system. In *Proceedings ACM-SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[113] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical Report CMU-CS-94-125, School of Computer Science, Carnegie Mellon University, March 7, 1994.

[114] Michael Stonebraker and Greg Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.

[115] Gilbert Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, Wellesley, MA, 1986.

[116] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, second edition, 1991.

[117] Chuanguang Sun. Efficient parallel solutions of large spare spd systems on distributed-memory multiprocessors. Technical Report CTC92TR102, Advanced Computing Research Institute, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY, August 1992.

[118] Chuanguang Sun. Parallel sparse orthogonal factorization on distributed-memory multiprocessors. Technical Report CTC93TR162, Advanced Computing Research Institute, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY, December 1993. (Revised, To appear in SIAM Journal of Scientific Computing).

[119] Chuanguang Sun. Parallel multifrontal solution of sparse linear least squares problems on distributed-memory multiprocessors. Technical Report CTC94TR185, Advanced Computing Research Institute, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY, July 1994.

[120] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1 and 2. Computer Science Press, Rockville, MD, 1988.

[121] Michael Wolfe. *High Performance Compilers*. Addison-Wesley, Redwood City, CA, 1996.

[122] Janet Wu, Raja Das, Joel Saltz, Harry Berryman, and Seema Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44(6), 1995.

[123] Richard Zippel. A constraint based scientific programming language. In *Principles and Practices of Constraint Programming*, chapter 7. MIT Press, 1995.