WATCH OUT FOR. . . WHAT?: MONITORING AND UNCERTAINTY IN SCIENTIFIC COMPUTING

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

Oliver Kennedy

May 2011

WATCH OUT FOR... WHAT?: MONITORING AND UNCERTAINTY IN SCIENTIFIC COMPUTING

Oliver Kennedy, Ph.D.

Cornell University 2011

As the amount of data involved in scientific research continues to grow, the need for powerful tools for organizing and analyzing this data grows with it. Despite considerable progress in this area by the database research community, the uptake of database technologies within the scientific community has been slow. Contributing to this limited adoption is a tendency to try to build complex, monolithic, total solution-systems, for a community that can rarely afford the resources to tie their existing infrastructures into such a system. This thesis explores two different directions for creating simpler, smaller, more general-purpose tools for doing data-processing in a scientific computing environment.

Grey-Box Probabilistic Databases are an attempt to create a general purpose tool for *efficiently* integrating database systems with an organization's existing model-building pipelines. By providing a pay-as-you-go approach to the trade-off between efficiency and integration effort, users can choose how much of their resources to commit as their needs develop.

Dynamic Data Management Systems are a new approach to building data processing systems. Instead of a monolithic data-processing infrastructure that typically includes (and has the performance penalties of supporting) functionality that the user does not require, a Dynamic Data Management System constructs entire data-management systems designed specifically to meet the requirements of the user's application.

BIOGRAPHICAL SKETCH

Ever since getting some books on BASIC programming in elementary school, Oliver Kennedy was hooked on computers. After graduating from Stuyvesant High School in New York City, Oliver joined the NYU Engineering program. While there, Oliver participated in research projects covering subjects from distributed systems security, to data visualization and education.

Immediately after graduating with honors, with a BS/BE in Computer Science/Computer Engineering, Oliver moved to sunny Ithaca, NY to join Cornell University's Computer Science PhD program. Despite a rocky start, he found a place for himself in the Big Red Data group, where he began working with Christoph Koch on supporting scientific computing through data management.

In his spare time Oliver plays with pretend sharp things, learning the intricate art of Stage Combat. Trained in the safe performance and instruction of Rapier and Rapier and Dagger, Oliver has achieved the rank of Advanced Scholar at Cornell's chapter of the Ring of Steel, where he regularly teaches, choreographs fights, and performs in such plays as "Zombies vs Shakespeare"[60].

To Dziadek, who would always make sure I was using the right tool for the job.

ACKNOWLEDGEMENTS

I would first like to thank the many Chrises in my life for making it that much more interesting. In particular, I would like to call out:

- C.Koch: For many years of advice, belief, patience, support, and not using his karate powers or nerf guns on me.
- C.Rockwell: For showing me a world of birds [75], lizards, and adventure.

In no particular order, I'd also like to thank:

- Yanif for always being there as a sounding board for my crazy ideas.
- Everyone in the Cornell Big Red Data group, the EPFL DATA and DIAS groups, as well as the denizens of Upson 331, for always having a good answer.
- Cornell's Becky, Kelly, and Stephanie, as well as EPFL's Simone and Christine, who's organizational superpowers have kept me from going insane many times over.
- Misters Kramer, Zamansky, Clancy, and Wong, without whom I would not be the programmer I am today.
- The Bather folk: Woody, Vijay, Eric, Tracy, Larry, Hesh, Sean, Christian, Michelle, and Nick, for exposing me to real research for the first time.
- The HCG folk: Gary, Lorna, Matt, Eduardo, Nicola, Carlos, Andrew, Antje, et al., for putting me on the right track with both conferences and relational databases.
- The Yahoo folk: Jay, Eric, Chad, John, Jimmy and Deepak, for a surprisingly fun and interesting summer working with diffy-qs.
- The Microsoft folk: Suman, Charles, Steve, Slawek, Pavel, and John, for giving me a much-needed splash of realism.
- The rest of my committee: Andrew, Jim, and Nate, for the advice, and for putting up with all the paperwork and my intercontinental issues.
- Ring. For the braaains.

Most importantly, I'd like to thank my family – especially my mom – for always being there for me, for always encouraging me to explore and understand, and for kicking my butt whenever I was slacking off.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . vi
List of Figures . . . ix

1 Introduction **1**
1.1 Grey-Box Probabilistic Databases . . . 3
1.1.1 Probabilistic Database Systems . . . 4
1.1.2 Black-Box Distributions . . . 5
1.1.3 Grey-Box Distributions . . . 6
1.1.4 Fingerprints and Canonical Distributions . . . 8
1.2 Dynamic Data Management Systems . . . 10
1.2.1 Large-Scale Data Analytics – But not as a Batch Job . . . 11
1.2.2 Database Monitoring . . . 13

2 Related Work **17**
2.1 Uncertainty in Data . . . 17
2.2 Stochastic Black-Boxes . . . 20
2.3 Dynamic Data Management Systems . . . 20

3 Pip **23**
3.1 C-Tables . . . 23
3.1.1 Relational algebra on c-tables . . . 24
3.1.2 Probabilistic C-Tables and Expectations . . . 26
3.1.3 Counting and Group-By . . . 27
3.2 Design of the PIP System . . . 28
3.2.1 Symbolic Representation . . . 29
3.2.2 Random Variables . . . 31
3.2.3 Condition Inconsistency . . . 34
3.2.4 Grey-Box Distributions . . . 36
3.3 Sampling and Integration . . . 38
3.3.1 Sampling Techniques . . . 39
3.3.2 Row-Level Sampling Operators . . . 41
3.3.3 Aggregate Sampling Operators . . . 45
3.4 Implementation . . . 47
3.4.1 Query Rewriting . . . 48
3.4.2 Defining Distributions . . . 51
3.4.3 Sampling Functionality . . . 52
3.5 Evaluation . . . 54

4 Jigsaw **62**
4.1 Simulating Business Scenarios . . . 62
4.1.1 Jigsaw Simulation Process . . . 67
4.1.2 Jigsaw Architecture . . . 69
4.1.3 Jigsaw Challenges . . . 70
4.2 Fingerprints . . . 72
4.2.1 Computing Fingerprints . . . 73
4.2.2 Indexing Fingerprints . . . 79
4.3 Markovian Jumps . . . 81
4.3.1 Fingerprinting Markov Processes . . . 83
4.3.2 Generating Estimator Functions . . . 86
4.4 Interactive What-ifs . . . 87
4.5 Evaluation . . . 91
4.5.1 Comparison of Two Prototypes . . . 92
4.5.2 Baseline Performance . . . 92
4.5.3 Indexing . . . 98
4.5.4 Markovian Jumps . . . 98

5 DBToaster **100**
5.1 The Architecture of a DDMS . . . 100
5.1.1 Application Interfaces . . . 102
5.1.2 DDMS Internals . . . 105
5.2 Realizing Agile Views . . . 107
5.2.1 View Maintenance in DBToaster . . . 107
5.2.2 Agile Views . . . 110
5.3 The DBToaster Compiler . . . 117
5.3.1 The DBToaster Workflow . . . 122
5.3.2 Simplifying Calculus Expressions . . . 131
5.3.3 Compiling Calculus Expressions . . . 134
5.3.4 Optimizing K3 . . . 137
5.4 Managing Storage in DBToaster . . . 145
5.4.1 Data Structures . . . 145
5.4.2 Partitioning and Co-clustering . . . 147
5.5 DBToaster in the Cloud . . . 151
5.5.1 Distributed Execution . . . 152
5.5.2 Execution Models . . . 153
5.5.3 Out-of-Order Processing . . . 156
5.5.4 Partitioning Schemes . . . 159
5.6 Evaluation . . . 160

6 Conclusions **164**

A PIP Query Q3 **167**

B A DBToaster Workflow Example **171**

Bibliography **178**

LIST OF FIGURES

1.1 An example relation in PIP. 7

3.1 Relational algebra on c-tables. 25
3.2 The Pip Query Engine Architecture 29
3.3 The PIP Postgres plugin architecture 48
3.4 Internal representation of C-Tables 51
3.5 Time to complete a 1000 sample query, accounting for selectivity-induced loss of accuracy. 54
3.6 Query evaluation times in PIP and Sample-First for a range of queries. Sample-First's sample-count has been adjusted to match PIP's accuracy. 55
3.7 RMS error across the results of 30 trials of (a) a simple group-by query Q_4 with a selectivity of 0.005, and (b) a complex selection query Q_5 with an average selectivity of 0.05. 56
3.8 Sample-First error as a fraction of the correct result in a danger-estimation query on the NSIDC's Iceberg Sighting Database. PIP was able to obtain an exact result. 61

4.1 An example Jigsaw query. 64
4.2 Example output of the `CapacityModel` with two purchase dates. 65
4.3 Jigsaw's interactive interface. 66
4.4 Processing optimization queries in Jigsaw 68
4.5 Jigsaw's Architecture 69
4.6 The use of identical seed values exposes simple correlations between output distributions for different inputs; When graphing the seed id against function output, such correlations appear as similarly shaped curves. 76
4.7 An example execution of the Markov Jump algorithm. 85
4.8 A Jigsaw query with a Markovian dependency 87
4.9 Black boxes used to evaluate Jigsaw 90
4.10 User Interface Wrapper vs Core Engine Simulator Timing comparison. Values are in time per parameter combination. 93
4.11 Jigsaw vs fully exploring the parameter space. 93
4.12 Computation time versus the size of structures in the Capacity model 94
4.13 Indexing in a static parameter space. Note the range on the y-axis – from 0.8 to 1.2 times the relative performance of naive array-scan. 95
4.14 Indexing, growing the parameter space with basis size. 96
4.15 Performance for a Markov process. 97

5.1 Dynamic Data Management System (DDMS) and Application Interface Architecture 101

5.2 Recursive query compilation in DBToaster. For query q, we produce a sequence of materializations and delta queries for maintenance: $\langle m, q' \rangle, \langle m', q'' \rangle, \langle m'', q''' \rangle$. This is a partial compilation trace, our algorithm considers all permutations of updates. . . . 111
5.3 Trigger functions generated by DBToaster for the query q, for all possible insertion orderings. The path taken through the compilation algorithm is expressed as part of the map name as seen for m_c and m_{cl} in the example walkthrough. 114
5.4 The DBToaster relational calculus 122
5.5 The DBToaster compilation workflow 123
5.6 The two-layer recursive structure of a DBToaster incremental plan. 127
5.7 The DBToaster relational calculus delta (δ) rewrite rules (based on [55]). ϕ is any comparison operator ($=, <, >, \neq, \leq, \geq$). 0 is the empty null schema relation. 136
5.8 An example of a messaging graph based on Example 5.2.1's query q (a) The messaging graph for the `on_insert_customer` event. (b) The effects of splitting view m_c on `ordkey`. (c) The effects of splitting m_c on `custkey`. 149
5.9 Supplemental data structures used to facilitate speculative execution in a distributed DDMS. 156
5.10 Tradeoff between Batch Processing and Incremental Computation on VWAP 162
5.11 Time to complete the VWAP query relative to the number of requests posed over the dataset 162
5.12 Tradeoff between Batch Processing and Incremental Computation on the star schema benchmark 163
5.13 Time to complete the star schema query relative to the number of requests posed over the dataset 163

A.1 A dataflow diagram of query 3. Dotted lines represent probabilistic data. 169

CHAPTER 1

INTRODUCTION

The incorporation of computers into the scientific process has brought about a wealth of discoveries. Computers automate, optimize, and parallelize the analysis of vast amounts of data, freeing scientists to focus on interpreting and understanding the results of the otherwise time-consuming analytical computations. Central to the analytical process is the scientist's ability to express their goals – typically achieved by writing computer programs (e.g., in languages like C [53], Python [64] or R [3]).

Fully-featured programming languages allow an extensive, and practically unlimited range of analytical options. However, the scientist is forced to consider not only their analytical goals, but to understand the wide range of algorithms which can be used to implement those goals and their related tradeoffs. Even relatively minor inefficiencies in the choice of algorithm can lead to substantial amounts of wasted time, as computations can last hours, days, or even weeks.

A primary goal of database research has been the automation of precisely this sort of algorithm selection process. However, despite considerable progress in this area, the uptake of database technologies within the scientific community has been slow. Contributing to this limited adoption is a tendency by the database community to follow a monolithic system-building process. By constructing systems that take full control of data storage, management and processing, the database designer imposes his (ultimately) limited view of how the data should be be stored and processed onto the user. In a field where the vast majority of data processing tasks involve corner cases, this tightly-controlled

approach to data management is of very limited use. Lacking better tools from the database community, the scientific community has turned to more simplistic, general-purpose systems (e.g. Hadoop [14]/Dryad [44]/MapReduce [26], PNuts [23]/Bigtable [20]/Cassandra [57], etc.), which ultimately result in less efficient data processing workflows.

An ideal solution sits between these two design extremes: Such tools provide simple, generalizable, and most importantly useable functionality for scientific research – but must also approach data management tasks in a principled way, making the most efficient use of (potentially limited) computational resources and the scientist's time.

This thesis explores several points in the design space of scientific computing tools: (1) Grey-Box Probabilistic Databases (GB-PDB) and (2) Dynamic Data Management Systems (DDMS). GB-PDBs allow scientists to leverage existing database and data management techniques for the efficient analysis of arbitrary uncertain data (e.g., models custom-designed by a scientist), even in large volumes. DDMS automate the process of generating efficient code for (potentially complex) data monitoring tasks, especially of high volume data streams.

Concretely, I present three systems, developed over the course of my studies: **PIP** [50] is a GB-PDB, which takes imperatively defined models and allows users to declaratively specify and query cross-model interactions. **Jigsaw** [52, 51] extends GB-PDB techniques by adding tools and techniques for parameter exploration and optimization. **DBToaster** [49] is a compiler for building DDMSs from SQL queries.

1.1 Grey-Box Probabilistic Databases

Uncertain data comes in many forms: Statistical models, scientific applications, and data extraction from unstructured text are all forms of uncertain data. Measurements have error margins while model predictions are often drawn from well known distributions. Traditional database management systems (DBMS) are ill-equipped to manage this kind of uncertainty.

Selecting a Shipping Provider. For example, a query may combine a model predicting per-customer profits with a model for predicting dissatisfied customers, perhaps as a result of a corporate decision to use a cheaper, but slower shipping company. Thus, the user might pose a query over this set of models, asking for profit loss due to dissatisfied customers.

The user might use a DBMS to store input parameters for these models (e.g., statistical metrics of historical customer satisfaction and shipping times). However, arbitrary queries made on the predictions do not translate naturally into queries on the corresponding model parameters: For example, a user who wishes to compute an expectation of the profit lost to customers dissatisfied with delivery times must first obtain a closed form solution by hand – assuming that a closed form solution even exists.

Furthermore, as the query asks for profit loss due to dissatisfied customers, the database need only consider profit from customers under those conditions where the customer is dissatisfied (i.e., the underlying model may include a correlation between ordering patterns and dependence on fast shipping). A query engine can identify and take advantage of such data-dependencies more efficiently than a human.

Enterprise Cluster Provisioning. Enterprises often need to evaluate business scenarios to assess and manage financial, engineering, and operational risks arising from uncertain data. Collaborations with a Microsoft Windows Azure cloud platform analytics team have revealed an increasing need for tools for developing timely plans for the expansion, deployment, and allocation of resources.

When making these plans, which can involve the allocation of millions of dollars, accurate and efficient simulation of many different business scenarios is critical to establish the validity of specific decisions in a timely manner.

Consider an analyst who wants to forecast the risk of running out of processing capacity in a cloud cluster. For that, she needs to combine various predictive models for CPU core demands and availability. These models are inherently uncertain due to imprecise predictions of future workload, possible downtime, delays in deployment, etc. Without effective tools, simulating and evaluating business scenarios based on uncertain models can be extremely challenging.

1.1.1 Probabilistic Database Systems

Probabilistic database management systems (PDBs) [21, 25, 91, 27, 10, 79, 80, 46, 88] aim at providing better support for querying uncertain data. Queries in these systems preserve the statistical properties of the data being queried, transforming the uncertain input data into a distribution over all possible query results – typically presented to the user as a metric (e.g., an expectation), or simplified representation (e.g., a histogram approximation of the probability density function, or PDF).

Probabilistic databases easily lend themselves to use as a sort of model-building and evaluation environment. Expressing a statistical model in the declarative language of probabilistic databases makes it more efficient to query the model's outputs, and allows the database to automatically generate an efficient evaluation strategy tailored to the query being asked.

The previously mentioned risk-management applications, built on top of a probabilistic database, could use the database itself to obtain error bounds on the results of arbitrary queries over its predictions. By encoding the statistical model for its predictions in the database itself, the risk-management application could even use the probabilistic database to estimate complex functions over many correlated variables in its model. In effect, the application could compute all of its predictions within the probabilistic database in the first place.

Evaluating queries over arbitrary distributions is computationally infeasible. Even simple queries over normal distributions can require the evaluation of integrals with no closed-form solution. Consequently, such systems resort to approximating the correct answer via Monte-Carlo style sampling techniques when no closed-form solution is available.

1.1.2 Black-Box Distributions

In an attempt to maximize use of closed-form solutions and/or exploit distribution characteristics, the vast majority of PDBs available restrict themselves to finite discrete distributions or certain well known (e.g., Gaussian, Poisson, etc.) continuous distributions.

At the other end of the spectrum is a recent trend towards a pure sampling-based approach. So-called variable generating, or VG-Functions [46] are user-provided black-box functions, which define a distribution by providing a means to generate samples from that distribution. For example, a user could define the gaussian distribution by providing a VG-Function that implements the Box-Muller algorithm [15].

Black-box sample generation can be used to integrate virtually any distribution into a probabilistic database framework.

Moreover, support for user-provided distributions is an extremely useful feature for the analysts, who derive their baseline models using specialized, external tools such as R [3]. Sparse, incomplete, or noisy data can transform even conceptually straightforward tasks (e.g., extracting the rate and volatility of demand growth) into daunting challenges that necessitate the use of such external tools.

Unfortunately, this brute-force approach to statistical analysis is dramatically less efficient than closed-form distribution-specific approaches.

1.1.3 Grey-Box Distributions

Although both efficiency and breadth are desirable characteristics, each user's requirements will be different; it can be entirely reasonable to accept inefficiency in a complex distribution that appears infrequently in user queries, as opposed to a simpler, far more common distribution.

Grey-Box Distributions (GBDs) provide a middle ground between these two

R	A	B
	1	$[Normal(\mu = 1, \sigma = 4)]$
	2	$[Normal(\mu = -1, \sigma = 6)]$
	3	$[Normal(\mu = 2, \sigma = 3)]$

Figure 1.1: An example relation in PIP.

extremes. At the core of a GBD is a black-box sample generation function – essentially a VG-Function. This minimal interface ensures: (1) The broadest range of distributions is supported, and (2) a low barrier for entry for users who wish to specify their own distributions.

GBDs can be extended for efficiency with optional user-provided and computer-generated metadata. Using this metadata the PDB engine can improve query performance by using closed-form solutions, exploiting analytical properties, or performing more directed sample-generation.

In Chapter 3, I present PIP, a GBD-based probabilistic database built on top of Postgres[2]. PIP processes queries over probabilistic data symbolically – the output of a query in PIP is a symbolic representation of the output table's distribution. When presenting this distribution to a user as an expectation, PDF, etc..., PIP employs a range of techniques to exploit metadata provided by GBD authors to avoid or limit sampling requirements.

Example 1.1.1 *Consider a table in a PDB such as the one in Figure 1.1.1. In this example table, column A is an integer and column B contains probabilistic data, here taking the form of multiple instantiations of the Normal distribution.*

Now consider computing the expectation of the following query:

```
SELECT SUM(B) FROM R WHERE B > 5
```

Even this simple query benefits greatly from having metadata about the distribution being queried. A purely sample-based approach will be forced to discard the majority of samples generated[1] *due to the sampled value not fulfilling the query constraint. Thus, over 5 times as many samples will need to be generated to obtain equally accurate results.*

Knowing that the Normal distribution has an easily-computable cumulative density function (CDF) allows PIP to avoid generating samples entirely – the CDF computes the expectation of the distribution exactly. Thus, when the Normal distribution is encoded as a GBD, the developer can include an optional function to compute the CDF of a Normal distribution with specified parameters. The query will return a value even without this optional function, but by providing it the user can make query evaluation more efficient – it is the user's decision whether the added efficiency is worth the added effort of providing PIP with a CDF.

1.1.4 Fingerprints and Canonical Distributions

Although user-provided metadata can be used to great effect, it is sometimes impractical to ask users to provide certain properties of a distribution. One such property is the relationship between the distribution and its input parameters. For example, the Normal distribution's parameters can be expressed as algebraic manipulations of a single canonical parametrization ($Normal(\mu, \sigma)$ =

[1] About 84% in this example, as the query constraint is precisely 1 standard deviation over the mean for each row of the example table

$Normal(0, 1) * \sigma^2 + \mu$). Such information is useful when processing queries, especially for queries where the PDB must explore a large parameter space to optimize for a given goal.

The corporate analyst considering the best time to purchase additional hardware for a computational cluster might use a CPU core availability model that accepts a set of candidate purchase dates and apply them according to a model for how long it takes to bring the hardware online. The analyst can then identify purchase dates that minimize the cloud's cost of ownership, given a bound on the risk of overload.

This is essentially a constrained optimization problem, albeit one where each iteration is an entire PDB subquery – each of which can independently take minutes or even hours to run. If a single (or a small number of) canonical parameterization(s) are associated with the CPU core availability model GBD, the underlying database can choose to sample the distribution only under canonical parameterizations and re-use the computed statistical metrics across the entire query.

Note that these sorts of correlations might often be obvious to a human onlooker. For example, excepting the days immediately after a hardware purchase, the day-by-day output of a simple CPU core availability model may be built out of the same distribution. However, forcing function authors to express such correlations through metadata is undesirable, as it negates the generality and clean abstraction offered by VG-functions: Data-dependent corner cases, discontinuities, and Markovian dependencies can make the metadata describing the correlations just as, or even more complex than the sample generation function itself.

In Chapter 4, I present Jigsaw, a PDB-based simulation framework that automatically extends a GBD by recording information about correlations between different parameterizations. It does this by using *fingerprints* of stochastic functions. The fingerprint of a stochastic black box function is a concise and easily-computable data structure that summarizes its output distribution. Thus, a fingerprint can be used to efficiently determine a sample generation function's correlation with another such function, or its own instantiations under different parameter values.

After such correlation has been detected, Jigsaw avoids expensive Monte Carlo estimation (and the associated function invocations) for a target point in the parameter space by using outputs for an already-explored, correlated point. The specific fingerprinting technique used is based loosely on random testing [38], a well known technique in software engineering: the fingerprint of a parameterized stochastic function is simply a sequence of its outputs under a fixed sequence of random inputs (i.e., seed of its pseudorandom number generator). The use of a fixed set of random seeds ensures a deterministic relationship between correlated outputs of the stochastic functions.

1.2 Dynamic Data Management Systems

The complexities, volumes, and rates of data used in scientific applications are growing more rapidly than ever before [40]. Large, rapidly changing datasets are commonly found in applications analyzing everything from stock-market transactions, to datacenter networks, to Twitter feeds, to earthquake-monitoring sensors.

Unfortunately, modern data management systems have not kept pace with this growth, treating updates and their impact on datasets and queries as an afterthought by extending DBMS with triggers and heavyweight views [34, 13, 93, 94], or only handling small, recent sets of records in data stream processing [4, 16, 68, 19].

This state of affairs is unacceptable, if the large-scale data processing challenges of the future are to be met. A new class of *Dynamic* Data Management Systems (DDMS) must be created to support not only stateful, complex data-processing tasks, but also be able to do so efficiently and incrementally.

The DDMS concept is a complete rethinking of data management techniques: This idea begins with the user interface, where rather than ad-hoc queries presented at runtime, persistent, application-specific data-processing tasks are pre-registered prior to the arrival of data. The complete redesign continues through to the implementation, where queries are processed as incrementally as possible, creating a range of possibilities for reducing processing requirements by increasing memory usage.

1.2.1 Large-Scale Data Analytics – But not as a Batch Job

Large-scale data analytics in the cloud are mostly performed on massively parallel processing engines such as map/reduce. These systems are not databases, as some members of the systems, scientific computing, and large-scale Web applications communities find important to emphasize. Nevertheless, the database community can play an important role in making such systems more useful and effective at posing *queries* over data.

Map/reduce-like systems achieve scalability at the cost of response time and interactivity. However, there is an increasing number of important applications of large-scale analytics that call for more interactivity or response times permitting online use. Among large Web applications, examples include (social or other) network monitoring and statistics [70], search with interactive feedback [12], interactive recommendations, keeping personalized Web pages at social networking sites up to date [29], and so forth. Many of these applications are not yet mission-critical to Web applications companies, but are becoming increasingly necessary for establishing and maintaining a competitive advantage.

Large-scale data analytics is equally present in more classical business applications such as data warehousing and scientific applications. Take the case of data warehousing with real-time updates: as data warehouses become increasingly mission-critical to commercial and scientific enterprises, the importance of up-to-date analyses increases. Traditionally, OLAP systems are not optimized for frequent updating, and may be considerably out-of-date. A DDMS dramatically improves the freshness of warehoused data.

A DDMS is well-suited for use in large-scale data analytics through its provision of large dynamic data structures as views, instead of forcing programmers to re-implement view computations manually on top of key-value stores. It emphasizes simple lightweight systems, and not monolithic DBMS engines. Continually fresh DDMS views may seem at odds with the bulk update processing dogma of large scale analytics systems, but enable important applications that require interactivity or event processing.

1.2.2 Database Monitoring

There is an ever-increasing set of use cases in which aggregate views over large databases need to be continuously maintained and monitored as the database evolves. These queries can be thought of as continuous queries on the stream of updates to a database. However, it is only moderately helpful to view this as a stream processing scenario since the queries depend on very large state (the database) rather than a small window of an update stream – such queries cannot be handled by data stream processing systems.

Examples include policy monitoring (e.g., to comply with regulatory requirements to monitor databases of financial institutions, say to detect money laundering schemes) [11], network security monitoring, aiming to detect sophisticated attacks that span extended time periods, and online data-driven simulations.

Probabilistic Databases. This last example is of particular interest – query processing in a probabilistic database is essentially a data-driven simulation: Each uncertain datum in the database can be thought of as a stream of random data selected according to its probability distribution, while the query can be thought of as a view over the database and these streams. As data streams in, the view is re-evaluated and statistics on its results are collected.

A similar approach [90] is to use Markov Chain Monte Carlo to repeatedly draw samples of the entire database (effectively treating it as an extremely high-dimensional random variable) by altering its state, one variable at a time. The results of the computation, which are incrementally maintained in a view, are sampled after each variable is updated.

The key technical database problem in both instances is to compute the view for as many samples as possible, as quickly as possible . This is precisely the kind of workload that DDMS are designed for.

Algorithmic trading with order books. Further examples of database monitoring can be found in certain forms of automated trading.

In recent years, algorithmic trading systems have come to account for a majority of volume traded at the major US and European financial markets (for instance, for 73% of all US equity trading volume in the first quarter of 2009 [42]). The success of automated trading systems depends critically on strategy processing speeds: trading systems that react faster to market events tend to make money at the cost of slower systems. Unsurprisingly, algorithmic trading has become a substantial source of business for the IT industry; for instance, it is the leading vertical among the customer bases for high-speed switch manufacturers (e.g., Arista [87]) and data stream processing.

A typical algorithmic trading system is run by mathematicians who develop trading strategies and by programmers and systems experts who implement these strategies to perform fast enough, using mainly low-level programming languages such as C [53]. Developing trading strategies requires a feedback loop of simulation, back-testing with historical data, and strategy refinement based on the insights gained. This loop, and the considerable amount of low-level programming that it causes, is the root of a very costly productivity bottleneck; the number of programmers often exceeds the number of strategy designers by an order of magnitude.

Trading algorithms often perform a considerable amount of data crunching

that could in principle be implemented as SQL views, but cannot be achieved by DBMS or data stream processing systems today: DBMS are not able to (1) update their views at the required rates (for popular stocks, hundreds of orders per second may be executed, even outside burst times) and stream engines are not able to (2) maintain large enough data state and support suitable query languages (non-windowed SQL aggregates) on this state. A data management system fulfilling these two requirements would yield a very substantial productivity increase that can be directly monetized – the holy grail of algorithmic trading.

To understand the need to maintain and query a large data state, note that many stock exchanges provide a detailed view of the market microstructure through complete bid and ask *limit order books*. The bid order book is a table of purchase offers with their prices and volumes, and correspondingly the ask order book indicates investors' selling orders. Exchanges execute trades by matching bids and asks by price and favoring earlier timestamps. Investors continually add, modify or withdraw limit orders, thus one may view order books as relational tables subject to high update volumes. The availability of order book data has provided substantial opportunities for automatic algorithmic trading.

Example 1.2.1 *To illustrate a specific DDMS application, consider a simple algorithmic trading strategy known as Static Order Book Imbalance (SOBI). SOBI computes a volume-weighted average price (VWAP) over those orders whose volume makes up a fixed upper k-fraction of the total stock volume in both bid and ask order books. SOBI then compares the two VWAPs and, based on this, predicts a future price drift (for example a bid VWAP larger than an ask VWAP indicates demand exceeds supply, and prices may rise). For simplicity, VWAP is presented for the bids only:*

```
SELECT AVG(b2.price * b2.volume) AS bid_vwap
  FROM bids b2
 WHERE k * (SELECT sum(volume) FROM bids)
         > (SELECT sum(volume) FROM bids b1
             WHERE b1.price > b2.price);
```

Focusing on the k-fraction of the order book closest to the current price makes the SOBI strategy less prone to attacks known as axes *(large tactical orders far from the current price that will thus not be executed but may confuse competing algorithms).*

Given continuously maintained views for VWAP queries on bid and ask order books, an implementation of the SOBI strategy only takes a few lines of code that trigger a buy or sell order whenever the ratio between the two VWAPs exceeds a certain threshold.

For trading algorithms to be successful, (1) views such as VWAP need to be maintained and monitored by the algorithms at or close to the trading rate. However, (2) the views cannot be expressed through time-, row- or punctuation-based window semantics. This lends weight to the need for DDMS that support agile views on large, long-lived state.

CHAPTER 2

RELATED WORK

2.1 Uncertainty in Data

The estimation of probabilities of continuous distributions frequently devolves into the computation of complex integrals. PIP's architecture allows it to identify cases where efficient algorithms exist to obtain a solution. For more complex problems not covered by these cases, PIP relies on Monte Carlo integration [66], a conceptually simple technique that allows for the (approximate) numerical integration of even the most general functions. Conceptually, to compute the expectation of function $q(\vec{x})$, one simply approximates the integral by taking n samples $\vec{x_1}, \ldots, \vec{x_n}$ for $\vec{X}$ from their distribution $p(\vec{X})$ and taking the average of the function evaluated on all n values.

In general, even taking a sample from a complicated PDF is difficult. Constraints imposed by queries break traditional Monte Carlo assumptions of normalization on $p(\vec{X})$ and require that the sampling technique account for them or lose precision. A variety of techniques exist to address this problem, from straightforward rejection sampling, where constraint-violating samples are repeatedly discarded, to more heavy duty Markov-chain Monte Carlo (MCMC, cf. e.g., [32]) style techniques such as the Metropolis-Hastings algorithm [67, 32].

Conditional tables (c-tables, [43]) are relational tables in which tuples have associated conditions expressed as boolean expressions over comparisons of random variables and constants. C-tables are a natural way to represent the *deterministic skeleton* of a probabilistic relational database in a succinct and tabular

form. That is, complete information about uncertain data is encoded using random variables, excluding only specifications of the joint probability distribution of the random variables themselves. This model allows representation of input databases with nontrivial statistical dependencies that are normally associated with graphical models – a key component of PIP's approach.

For *discrete* probabilistic databases, a canon of systems has been developed that essentially use c-tables, without referring to them as such. MystiQ [25] uses c-tables internally for query processing but uses a simpler model for input databases. Trio [91] uses c-tables with additional syntactic sugar and calls conditions *lineage*. MayBMS [8] uses a form of c-tables called U-relations that define how relational algebra representations of queries can encode the corresponding condition transformations.

ORION [80] is a probabilistic database management system for continuous distributions that can alternate between sampling and transforming distributions. However, their representation system is not based on c-tables but essentially on the world-set decompositions of [9], a factorization based approach related to graphical models. Selection queries in this model may require an exponential blow-up in the representation size, while selections are efficient in c-tables.

The MCDB system[46] has promoted an integrated sampling-based approach to probabilistic databases. Conceptually, MCDB uses a *sample-first* approach: it first computes samples of entire databases and then processes queries on these samples. This is a very general and flexible approach, largely due to its modular approach to probability distributions via black box sample generators called VG Functions. Using Tuple-Bundles, a highly compressed representation of the

sampled database instances, MCDB shares computation across instances where possible during query evaluation.

More recently, MCDB has been extended with techniques for highly selective queries [37], but still does not exploit any analytically useful properties of individual black-box functions.

Jigsaw builds on probabilistic database techniques – specifically those based on VG-Functions, like MCDB and PIP. By providing functionality for efficiently examining the output of VG-Functions across different parameters, Jigsaw enables the efficient use of probabilistic databases for parameter optimization and exploration tasks.

An area of database research related to probabilistic databases involves representing continuous functions as tables. Pulse [6], MauveDB [27], and FunctionDB [85] allow users to construct functional models within the database. Queries posed over these models are evaluated symbolically to the extent possible, substantially improving performance. However, these systems necessitate a functional representation of the data being modeled, and thus lack the generality of VG-Functions. However, once Jigsaw has extracted a set of mapping functions $\mathcal{M}$ for an entire parameter space, symbolic querying techniques such as these could be applied to improve performance further.

The integration of specialized modeling tools with database systems [92] has been explored. However, while such tools streamline the process of fitting a model, they do not focus on model evaluation. One could imagine such a tool being integrated into the Jigsaw workflow for the construction of VG-Functions and parameterizations thereof.

Constrained optimization has also been considered in the context of databases [31], but with a view towards minimizing IO requirements. Indexes containing data bounds are used to prune the search space. However, this approach relies on the continuity of the function being optimized; VG-Functions negate this assumption.

2.2 Stochastic Black-Boxes

Functional representation of stochastic black-boxes is a field that has been explored extensively. Techniques ranging from simple curve-fitting, wavelets [36], various space transforms [7], and even simple hat functions [28] have been proposed as mechanisms for producing functional representations of stochastic black-boxes.

A number of techniques [78, 56] have also been developed for performing optimization over black-box functions. However, all of these techniques are developed for uncontrollable black-boxes, typically real-world processes. Consequently, they are limited to a regression-style approach. Conversely, Jigsaw controls the source of randomness within the function, which allows it to deterministically generate fingerprints.

2.3 Dynamic Data Management Systems

Compared to a classical DBMS, a DDMS differs in its reaction to updates. To minimize response times, updates must be performed immediately upon ar-

rival, precluding bulk processing. This determines the programming model: compared to a DBMS, control flow is reversed, and the DDMS invokes application code, not vice versa.

An active DBMS [18] could simulate a DDMS through triggers, but is not optimized for such workloads, and even if support for state-of-the-art incremental view maintenance is present, performs very poorly. Thus, DDMS differ from active databases in their being optimized for different workloads. DDMS are optimized for event processing and monitoring tasks, while active database systems are optimized to support traditional DBMS functionality such as transactions, which are not necessarily present in DDMS.

Compared to a data stream processing system and particularly an event processing system (such as Cayuga [16], SASE+ [5]), DDMS have much larger states, which will usually have to be maintained in secondary storage, and require drastically different query processing techniques. In a stream processor, the queries reside in the system while the data streams by. In a DDMS on the other hand, the data state is maintained in the system while a stream of updates passes through (much more like an OLTP system).

Moreover, event and stream processors [4, 68, 19] support drastically different query languages which are designed to ensure that only very small state has to be maintained, using windows or constructs from formal language theory [89]. DDMS views are often rather complex and expensive, including large non-windowed joins and aggregation. In general, a DDMS can be expected to support standard SQL. The query processing techniques most suitable for such workloads come from DBMS research – incremental view maintenance in particular – and update stream research [30] but do not scale to high-frequency

view maintenance.

The recursive compilation process used in Agile Views bears a close resemblance to Automatic Differentiation techniques [73]. Certainly, the notion of differentiating computer programs has been considered before [48]. This resemblance is not coincidental – the `calc_t` representation used by DBToaster is a ring of sets and the delta operation can be thought of as a form of differentiation applied to sets. However, unlike arithmetic expressions (or computer programs reducible to arithmetic over functions with hardcoded differentials), DBToaster relational calculus expressions operate over sets, and incorporate a notion of information passing (e.g., via the product and definition operators) that must be accounted for in the delta rules.

CHAPTER 3

PIP

This chapter describes PIP, a probabilistic database system for continuous probability distributions. PIP is designed to provide end-users with both flexibility and scalability in terms of how uncertain data is specified. By offering a range of options for how uncertain data is defined, PIP allows users to scale the effort they are willing to commit to defining their uncertain data with respect to the amount of performance they require. Furthermore, PIP's symbolic representation of uncertainty allows it to defer evaluation of uncertain query results until after query processing is complete and the expression to be computed is fully known and can thus be evaluated as efficiently as possible.

PIP was developed in collaboration with my advisor Christoph Koch, and is based on MayBMS [10, 54], developed by Lyublena Antova, Jiewen Huang, Christoph Koch, and Dan Olteanu. PIP was originally published at ICDE 2009 [50].

This material is based upon work supported by the National Science Foundation under Grant IIS-0812272. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

3.1 C-Tables

A c-table over a set of variables is a relational table[1] extended by a column for holding a *local condition* for each tuple. A local condition is a Boolean combination (using "and", "or", and "not") of atomic conditions, which are constructed from variables and constants using $=$, $<$, $\leq$, $\neq$, $>$, and $\geq$. The fields of the remain-

[1] In the following, a multi-set semantics for tables is used: Tables may contain duplicate tuples. Set transformations are defined in comprehension notation $\{\!|\, \cdot \mid \cdot \,|\!\}$ with $\in$ as an iterator. Transformations preserve duplicates. $\uplus$ denotes bag union, which can be thought of as list concatenation if the multi-sets are represented as unsorted lists.

ing data columns may hold domain values or variables.

Given a variable assignment θ that maps each variable to a domain value and a condition ϕ, the notation $\theta(\phi)$ denotes the condition obtained from ϕ by replacing each variable X occurring in it by $\theta(X)$. Analogously, $\theta(\vec{t})$ denotes the tuple obtained from tuple $\vec{t}$ by replacing all variables using θ.

The semantics of c-tables are defined in terms of possible worlds as follows. A possible world is identified with a variable assignment θ. A relation R in that possible world is obtained from its c-table C_R as

$$R := \{\!\{ \theta(\vec{t}) \mid (\vec{t}, \phi) \in C_R, \theta(\phi) \text{ is true} \}\!\}.$$

That is, for each tuple $(\vec{t}, \phi)$ of the c-table, where ϕ is the local condition and $\vec{t}$ is the remainder of the tuple, $\theta(\vec{t})$ exists in the world if and only if $\theta(\phi)$ is true. Note that each c-table has at least one possible world, but worlds constructed from distinct variable assignments do not necessarily represent different database instances.

3.1.1 Relational algebra on c-tables

Evaluating relational algebra on c-tables (and without the slightest difference, on probabilistic c-tables, since probabilities need not be touched at all) is surprisingly straightforward. The evaluation of the operators of relational algebra on multi-set c-tables is summarized in Figure 3.1. An explicit operator "distinct" is used to perform duplicate elimination.

$$
\begin{aligned}
C_{\sigma_\psi(R)} &= \{\!\!\{ (\vec{r}, \phi \wedge \psi[\vec{r}]) \mid (\vec{r}, \phi) \in C_R \}\!\!\} \\
&\ldots\ \psi[\vec{r}] \text{ denotes } \psi \text{ with each reference to} \\
&\quad\ \text{a column } A \text{ of } R \text{ replaced by } \vec{r}.A. \\
C_{\pi_{\vec{A}}(R)} &= \{\!\!\{ (\vec{r}.\vec{A}, \phi) \mid (\vec{r}, \phi) \in C_R \}\!\!\} \\
C_{R \times S} &= \{\!\!\{ (\vec{r}, \vec{s}, \phi \wedge \psi) \mid (\vec{r}, \phi) \in C_R, (\vec{s}, \psi) \in C_S \}\!\!\} \\
C_{R \cup S} &= C_R \uplus C_S \\
C_{\text{distinct}(R)} &= \{\!\!\{ (\vec{r}, \bigvee \{\phi \mid (\vec{r}, \phi) \in C_R\}) \mid (\vec{r}, \cdot) \in C_R \}\!\!\} \\
C_{R-S} &= \{\!\!\{ (\vec{r}, \phi \wedge \psi) \mid (\vec{r}, \phi) \in C_{\text{distinct}(R)}, \\
&\qquad \text{if } (\vec{r}, \pi) \in C_{\text{distinct}(S)} \text{ then } \psi := \neg\pi \\
&\qquad \text{else } \psi := \text{true} \}\!\!\}
\end{aligned}
$$

Figure 3.1: Relational algebra on c-tables.

Example 3.1.1 *Suppose a database captures customer orders expected for the next quarter, including prices and destinations of shipment. The order prices are uncertain, but a probability distribution is assumed. The database also stores distributions of shipping durations for each location. Here are two c-tables defining such a probabilistic database:*

C_{Order}	*Cust*	*ShipTo*	*Price*	θ
	Joe	*NY*	X_1	*true*
	Bob	*LA*	X_3	*true*

C_{Order}	*Dest*	*Duration*	θ
	NY	X_2	*true*
	LA	X_4	*true*

Here, a suitable specification of the joint distribution p of the random variables $X_1, \ldots, X_4$ occurring in this database is assumed.

Consider the relational algebra query

$$\pi_{\text{Price}}(\sigma_{\text{ShipTo=Dest}}(\sigma_{\text{Cust}='Joe'}(\text{Order}) \times \sigma_{\text{Duration} \geq 7}(\text{Shipping}))).$$

Evaluating this query in stages:

$$C_{\sigma_{\text{Cust}='Joe'}(\text{Order})} = \{\!\!\{ ((Joe, NY, X_1), true) \}\!\!\}$$

$$C_{\sigma_{\text{Duration}\geq 7}(\text{Shipping})} = \{\!\{ ((NY, X_2), X_2 \geq 7), ((LA, X_4), X_4 \geq 7) \}\!\}$$

$$C_{\sigma_{\text{Cust}='Joe'}(\text{Order})\times\sigma_{\text{Duration}\geq 7}(\text{Shipping})} = \{\!\{ \quad ((Joe, NY, X_1, NY, X_2), X_2 \geq 7), \quad ((Joe, NY, X_1, LA, X_4), X_4 \geq 7) \quad \}\!\}$$

3.1.2 Probabilistic C-Tables and Expectations

A *probabilistic c-table* [33, 54] is a c-table in which each variable is simply considered a (discrete or continuous) *random variable*, and a joint probability distribution is given for the random variable. As a convention, discrete random variables are denoted by $\vec{X}$ and continuous random variables by $\vec{Y}$. Henceforth, it is always implicitly assumed that *discrete random variables have a finite domain*.

Assume a suitable function $p(\vec{X} = \vec{x}, \vec{Y} = \vec{y})$ specifying a joint distribution which is essentially a PDF on the continuous and a probability mass function on the discrete variables. To clarify this, p is such that the expectation of a function q can be defined as

$$E[q] = \sum_{\vec{x}} \int_{y_1} \cdots \int_{y_n} p(\vec{x}, \vec{y}) \cdot q(\vec{x}, \vec{y})\, d\vec{y} \approx \frac{1}{n} \cdot \sum_{i=1}^{n} q(\vec{x}_i, \vec{y}_i)$$

given samples $(\vec{x}_1, \vec{y}_1), \ldots, (\vec{x}_n, \vec{y}_n)$ from the distribution p.

Events (sets of possible worlds) are specified via Boolean conditions ϕ that are true on a possible world (given by assignment) θ iff the condition obtained by replacing each variable x occurring in ϕ by $\theta(x)$ is true. The characteristic function χ_ϕ of condition (event) ϕ returns 1 on a variable assignment if it makes ϕ true and returns zero otherwise. The probability $\Pr[\phi]$ of event ϕ is simply $E[\chi_\phi]$.

The expected sum of a function h applied to the tuples of a table R,

```
SELECT expected_sum(h(*)) FROM R;
```

can be computed as

$$E\Big[\sum_{\vec{t}\in R} h(\vec{t})\Big] = E\Big[\sum_{(t,\phi)\in C_R} \chi_\phi \cdot h(t)\Big] = \sum_{(t,\phi)\in C_R} E\Big[\chi_\phi \cdot (h \circ t)\Big]$$

(the latter by linearity of expectation).

Here $t(\vec{x}, \vec{y})$ denotes the tuple t, where any variable that may occur is replaced by the value assigned to it in $(\vec{x}, \vec{y})$.

Example 3.1.2 *Returning to the earlier example, for $C_R = \{\!\{(x_1, x_2 \geq 7)\}\!\}$, the expected sum of prices is*

$$\sum_{(t,\phi)\in C_R} \int_{x_1} \cdots \int_{x_4} p(\vec{x}) \cdot \chi_\phi(\vec{x}) \cdot t(\vec{x}).\text{Price}\; d\vec{y} = \int_{x_1} \cdots \int_{x_4} p(\vec{x}) \cdot \chi_{X_2 \geq 7}(\vec{x}) \cdot x_1 \; d\vec{y}.$$

3.1.3 Counting and Group-By

Expected count aggregates are special cases of expected `SUM` aggregates where h is a constant function 1. Grouping by (continuously) uncertain columns is of doubtful value – the probability of two continuous random variables being equal approaches zero. Grouping by non-probabilistic columns (i.e., which contain no random variables) poses no difficulty in the c-tables framework: the above summation simply proceeds within groups of tuples from C_R that agree on the group columns.

In particular, by delaying any sampling process until after the relational algebra part of the query has been evaluated on the c-table representation, we find

it easy to create as many samples as we need for each group in a goal-directed fashion. This is a considerable strong point of the c-tables approach used in PIP.

3.2 Design of the PIP System

Representing the uncertain components of a query's output symbolically as a c-table makes a variety of integration techniques available for use in evaluating the statistical characteristics of the expression. Consider an application or task that incorporates a set of independent input models, establishing correlations between them exclusively via queries. PIP can detect this lack of dependency, compute metrics over each model independently, and combine the results afterwards. Even with relatively straightforward integration techniques, this extra information (and information similarly obtained through runtime analysis of the symbolic representations) can be used to substantially improve query performance and/or accuracy.

Accuracy in particular, is relevant in cases where the integral has no closed form and exact methods are unavailable. This is the case in a surprising range of practical applications, even when strong simplifying assumptions are made about the input data.

Even if the input data contains only independent variables sampled from well-studied distributions (e.g., the normal distribution), it is still possible for queries to create complex statistical dependencies in their own right. It is well known, at least in the case of discrete and finite probability distributions, that relational algebra on block-independent-disjoint tables can construct any finite probability distribution [74, 43].

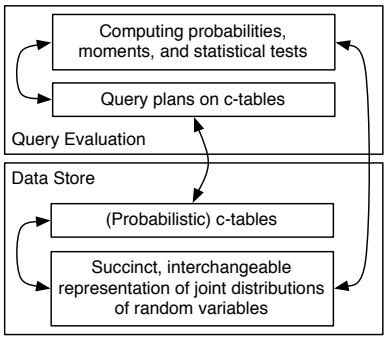


Figure 3.2: The Pip Query Engine Architecture

3.2.1 Symbolic Representation

PIP represents probabilistic data values symbolically, using random variables defined in terms of parametrized probability distribution classes called grey-box distributions (GBDs). PIP includes several commonly used distributions as GBDs (e.g., Normal, Uniform, Exponential, Poisson), and may be extended with more by end-users. Variables are treated as opaque while they are manipulated by traditional relational operators. The resulting symbolic representation is a c-table. As the final stage of the query processing pipeline, special expectation operators defined within PIP compute expectations and moments of the uncertain data, or sample the data to generate histograms. This process is illustrated in Figure 3.2.

These expectation operators are invoked with a lossless representation of the expression to be evaluated. Because the variables have been treated as opaque, the expectation operator can obtain information about the distribution a variable corresponds to. Similarly, the lossless representation allows on-the-spot generation of samples if necessary; There is no bias from samples shared between multiple query runs.

The primary component of a GBD encoding a distribution p is a sampling function that generates values sampled from p. GBD authors can (but need not) provide supplemental information (e.g., functions defining the PDF and the CDF) about distributions they extend PIP with. This additional information benefits end-users at query time – PIP uses it to accelerate the sampling process or potentially sidestep it entirely.

Example 3.2.1 *Consider a query that essentially computes the probability of a random variable sampled from a GBD falling within specified bounds. If the GBD specifies only a sampling function, then PIP is required to use Monte Carlo simulation to obtain an approximate result – something that involves hundreds, or even thousands of invocations of the sample generating function. If the GBD specifies a cumulative distribution function (CDF), PIP can obtain a precise result with only two invocations of the CDF.*

Because the symbolic representation PIP uses is lossless, intermediate query results or views may be materialized. Expectations of values in these views or subsequent queries based on them will not be biased by estimation errors introduced by materializing the view. This is especially useful when a significant fraction of query processing time is devoted to managing deterministic data (e.g., to obtain parameters for the model's variables). Not only does this enable the use of materialized views for commonly used subqueries, but it improves the efficiency of online sampling: the sampler does not need to evaluate the entire query from scratch to generate additional samples.

Example 3.2.2 *Recall Example 3.1.2. The result of the relational algebra part of the example query can be easily computed as*

R	*Price*	*Condition*
	Y_1	$Y_2 \geq 7$

without looking at p.

This c-table compactly represents all data still relevant after the application of the relational algebra part of the query, other than p, which remains unchanged. Sampling from R to compute

```
SELECT expected_sum(Price) FROM R;
```

is a much more focused effort.

First, only the random variables relating to Joe must be considered; but determining that random variable Y_2 is relevant while Y_4 is not requires executing a query involving a join. It is far more efficient to perform the query first, before generating samples of each variable.

Second, assume that delivery times are independent from sales volumes. Then the query result is approximated by first sampling an Y_2 value and only sampling an Y_1 value if $Y_2 \geq 7$. Otherwise, the Y_1 value is 0. If $Y_2 \geq 7$ is relatively rare (e.g., the average shipping times to NY are very slow, with a low variance), this may reduce the amount of samples for Y_1 that are first computed and then discarded without seeing use considerably. If CDFs are available, it is of course possible to do even better.

3.2.2 Random Variables

At the core of PIP's symbolic representation of uncertainty is the random variable. The simplest form of a random variable in PIP consists of a unique iden-

tifier, a subscript (for multi-variate distributions), a distribution class, and a set of parameters for the distribution.

For example, $[Y \Rightarrow Normal(\mu, \sigma^2)]$ represents a normally distributed random variable X with mean μ and standard deviation σ^2. Multivariate distributions are specified as arrays, like this multivariate normal distribution: $[Y[n] \Rightarrow MVNormal(\mu, \sigma^2, n)]$.

In summary, random variables are declared as a combination of distribution and a set of parameters for that distribution. The variable is automatically assigned a unique identifier to ensure that the sampling process generates consistent values for each appearance of the variable across the entire database.

The result of expressions over random variables is stored as an arithmetic formula tree, where leaves are random variables or constants and nodes are arithmetic operators. Because such equations themselves describe random variables, the terms equation and random variable will be used interchangeably. Also note that although PIP is currently limited to arithmetic operators, it is possible to use this same technique to support any non-recursive expression.

Example 3.2.3 *Equations of random variables can be combined freely with constant expressions, both in the target clause and where clauses of a select statement. For example:*

```
SELECT Y1 * 2 FROM R WHERE Y1 + Y2 < 2 + 4
```

All target expressions in the select statement with random variables (`Y1 * 2` *in the example above), are encoded into the output as an arithmetic formula tree. Where clause*

expressions with random variables (`Y1 + Y2 < 2 + 4` *above), are encoded into the output as part of the c-table condition.*

C-table conditions are encoded as a boolean formula of *atoms*, arbitrary inequalities of random variable expressions. The independent probability, or *confidence* of the tuple is the probability of the condition being satisfied.

Conjunctive Conditions. Recall that with the exception of duplicate elimination and negative relational algebra, queries over c-tables with exclusively conjunctive conditions produce exclusively conjunctive c-table conditions. Thus it makes sense to particularly optimize this scenario [8]. In the case of positive relational algebra with the duplicate elimination operator (i.e., duplicate elimination is traded against difference), the conditions can still be efficiently maintained in DNF, i.e., as a simple disjunction of conjunctions of atomic conditions.

Without loss of generality, the model can be limited to conditions that are conjunctions of constraint atoms. Generality is maintained by using bag semantics to encode disjunctions; Disjunctive terms are encoded as separate rows, and the distinct operator is used to coalesce terms. This restriction provides several benefits. First, constraint validation is simplified; A pairwise comparison of all atoms in the clause is sufficient to catch the inconsistencies listed above. As an additional benefit, if all atoms of a clause define convex and contiguous regions in the space $\vec{x}, \vec{y}$, these same properties are also shared by their intersection.

3.2.3 Condition Inconsistency

Conditions can become *inconsistent* when contradictory conditions are introduced conjunctively, which may happen in the implementations of the operators selection, product, and difference. If such tuples are discovered, they may be freely removed from the c-table.

A condition is consistent if there is a variable assignment that makes the condition true. For general boolean formulas, deciding consistency is computationally hard. But it is not necessary to decide it during the evaluation of relational algebra operations. Rather, straightforward cases of inconsistency are immediately removed to clean-up c-tables and reduce their sizes. The later Monte Carlo simulation phase enforces any remaining inconsistencies.

1. The consistency of conditions not involving variable values is always immediately apparent.
2. Conditions $X_i = c_1 \wedge X_i = c_2$ with constants $c_1 \neq c_2$ are always inconsistent.
3. Equality conditions over continuous variables $Y_j = (\cdot)$, with the exception of the identity $Y_j = Y_j$, are not inconsistent but can be treated as such (the probability mass will always be zero). Similarly, conditions $Y_j \neq (\cdot)$, with the exception of $Y_j \neq Y_j$, can be treated as true and removed or ignored.

With respect to (finite) discrete variables, inconsistency detection may be further simplified. Rather than storing where clause conditions over discrete variables symbolically in the c-table condition, the table may be expanded by enumerating all possible instantiations of each row. Expressed this way, c-table conditions over variables are limited to equality constraints of the form

Algorithm 1: `consistencyCheck`(C)

Require: A set of conditions C.

Ensure: false if the conditions are provably inconsistent, true otherwise.

1: **for all** *Continuous* variable group K {See Section 3.3.1} **do**
2: initialize bounds map S s.t. $S[X] = [-\infty, \infty]\ \forall X \in K$
3: **repeat**
4: **for all** Expression $E \in K$ **do**
5: **if** at most 1 variable in E is unbounded **then**
6: **for all** X {Use the bounded variables to shrink the variable bounds} **do**
7: $S[X] \leftarrow S[X] \cap \texttt{tighten}_{class(E)}(X, E, S)$ {A different `tighten` method is defined for different classes of expression (e.g., polynomial expressions of degree 1). This step is skipped if no `tighten` method is available for this expression.}
8: **if** $\exists X$ s.t. $S[X] = \emptyset$ **then**
9: **return** false
10: **until** fixedpoint of S
11: **return** true

$x_1 = constant$, expressing the value assigned that each variable in this instance of the row, and discrete variable columns may be treated as constants for the purpose of consistency checks. As shown in [8], deterministic database query optimizers do a satisfactory job of ensuring that constraints over discrete variables are filtered as soon as possible.

PIP's approach to consistency checking expressions of continuous variables involves producing a *bounds map*, the range of values that each variable being

Algorithm 2: $\texttt{tighten}_{poly1}(X, E, S)$

(An example variable bounds reducer for expressions with polynomial degree 1)

Require: A variable X, expression E, and a bounds map S

Ensure: A tighter bound on variable X if one is possible.

1: Express E in normal form $aX + bY + cZ + \ldots > 0$

2: **if** $a > 0$ **then**

3: **return** $[-(b \cdot \max(S[Y]) + c \cdot \max(S[Z]) + \ldots)/a, \infty]$

4: **else if** $a < 0$ **then**

5: **return** $[-\infty, -(b \cdot \max(S[Y]) + c \cdot \max(S[Z]) + \ldots)/a]$

sampled from can take. The bounds map is progressively tightened until a fixed point is reached, using each expression to propagate bounds on one variable through to others in the expression. If the bounds map contains a variable with an empty range (i.e., there is no value that the variable could possibly take), the expression is deemed inconsistent. This process is summarized as Algorithm 1, and a simple algorithm for tightening bounds on expressions of polynomial degree 1 is presented in Algorithm 2.

3.2.4 Grey-Box Distributions

PIP defines variables in terms of distribution classes – until it becomes necessary to compute metrics over a random variable or expression of random variables, PIP is agnostic to the details of how a variable is implemented. Isolating variable use to this single *sampling* component of the system makes it easy for users to

extend PIP with additional distribution classes.

Distributions in PIP are defined as a Black-Box sampling function which generates samples from the distribution (i.e., like a VG-Function [46]), plus a set of metadata associated with the distribution. The additional information in one such Grey-Box Distributions (GBDs) allows PIP to compute metrics over variables and variable expressions with greater efficiency. Concretely, PIP currently utilizes the following distribition-specific metadata (if it is available):

1. A function for efficiently computing the distribution's probability density function (PDF)
2. A function for efficiently computing the distribution's cumulative density function (CDF)
3. A function for efficiently computing the inverse cumulative density function (CDF^{-1})

Note that this specific selection of metadata has been selected primarily to demonstrate the potential of PIP's approach (i.e., symbolic execution and Grey-Box Distributions). Further distribution-specific values like weighted-sampling, mean, entropy, and the higher moments can be used by more advanced statistical methods to achieve even better performance. The process of defining a variable distribution is described further in Section 3.4.

Though PIP abstracts the details of a variable's distribution from query evaluation, it distinguishes between discrete and continuous distributions. As described in Section 3.1, existing research into c-tables has demonstrated efficient ways of querying variables sampled from discrete distributions. PIP employs similar techniques when it is possible to do so.

3.3 Sampling and Integration

Query evaluation in PIP occurs in two stages: Query and Sampling. PIP relies on Postgres' native engine to evaluate queries; As described in Section 3.1, a query rewriting pass suffices to translate c-tables relational algebra extensions into traditional relational algebra. Details on how query rewriting is implemented are provided in Section 3.4.

As the query is being evaluated, special sampling operators in the query are used to transform random variable expressions into histograms, expectations, and other statistical metrics. The computation of both metrics and probabilities reduces to numerical integration in the general case, and the dominant technique for doing this is Monte Carlo simulation. The approximate computation of expectation

$$E[\chi_\phi \cdot (h \circ t)] = \frac{1}{n} \cdot \sum_{i=1}^{n} p(\vec{y}_i) \cdot \chi_\phi(\vec{y}_i) \cdot h(t(\vec{y}_i)) \tag{3.1}$$

faces a number of difficulties. In particular, samples for which χ_ϕ is zero do not contribute to an expectation. If ϕ is a very selective condition, most samples do not contribute to the summation computation of the approximate expectation. (This is closely related to the most prominent problem in online aggregation systems [39, 77], and also in [46]).

Example 3.3.1 *Consider a row containing the variable*

$$[Y \Rightarrow Normal(\mu = 5, \sigma^2 = 10)]$$

and the condition predicate $(Y > -3)$ *and* $(Y < 2)$*. The expectation of the variable Y in the context of this row is not* 5*, but rather* ~ 0.17 *– it is based only on samples of Y that fall in the range* $(-3, 2)$*.*

3.3.1 Sampling Techniques

Rejection Sampling One straightforward approach to this problem is to perform rejection sampling; sample sets are repeatedly generated until a sufficient number of viable (satisfying) samples have been obtained.

However, without scaling the number of samples taken based on $E[\chi_\phi]$, information can get very sparse and the approximate expectations will have a high relative error. Unfortunately, as the probability of satisfying the constraint drops, the work required to produce a viable sample increases; More efficient mechanisms are necessary.

Sampling using inverse CDFs As an alternative to generator functions, PIP can also use the inverse-transform method [59]. If available, the distribution's inverse-CDF function is used to translate a uniform-random number in the range $[0, 1]$ to the variable's distribution.

This technique makes constrained sampling more efficient. If the uniform-random input is selected from the range $[CDF(a), CDF(b)]$, the generated value is guaranteed to fall in the range $[a, b]$. Even if precise constraints can not be obtained, this technique still reduces the volume of the sampling space, increasing the probability of getting a useful sample.

In the event that the inverse CDF is available, but the CDF is not, this technique is still useful. Instead of sampling from the range $[CDF(lower), CDF(upper)]$, PIP keeps track of the lowest sample value above the upper bound and the highest sample value below the lower bound and the corresponding input to the inverse CDF. PIP effectively learns the values of the input bounds over the course

of its normal sampling process.

If precise bounds can be derived from the constraints on a given variable, this process guarantees that each sample generated will satisfy the constraint. Even if only weak bounds are available, the process still provides a benefit. By reducing the size of the sampling area, the probability of selecting a viable sample is still increased.

Exploiting independence Prior to sampling, PIP subdivides constraint predicates into minimal independent subsets: sets of predicates that do not share common variables. When determining subset independence, variables generated by the same multivariate distribution are effectively treated as the same variable. For example, consider the one row c-table of nullary schema (i.e., there is only a condition column)

R	ϕ_2
	$(Y_1 > 4) \wedge ([Y_1 \cdot Y_2] > Y_3) \wedge (A < 6)$

In this case, the atoms $(Y_1 > 4)$ and $([Y_1 \cdot Y_2] > Y_3)$ form one minimal independent subset, while $(A < 6)$ forms another.

Because these subsets share no variables, each may be sampled independently. Sampling fewer variables at a time reduces the work lost generating non-satisfying samples, and decreases the frequency with which this happens.

Metropolis A final alternative available to PIP, is the Metropolis algorithm [67]. Starting from an arbitrary point within the sample space, this algorithm performs a random walk weighted towards regions with higher probability

densities. Samples taken at regular intervals during the random walk may be used as samples of the distribution.

The Metropolis algorithm has an expensive startup cost, as there is a lengthy [2] 'burn-in' period while it generates a sufficiently random initial value. Despite this startup cost, the algorithm typically requires only a relatively small number of steps between each sample. Consequently, the Metropolis algorithm is ideally suited for generating large numbers of samples when the CDF is not available and the probability of sampling a given value is small.

We can estimate the work required for both Metropolis and Naive rejection sampling.

$$W_{metropolis} = C_{burn\ in} + [\#\ samples] \cdot C_{steps\ per\ sample}$$

$$W_{naive} = \frac{1}{1 - P[reject]} \cdot [\#\ samples]$$

By generating a small number of samples for the subgroup, PIP can generate a rough estimate of $P[reject]$ and decide which approach is less expensive.

3.3.2 Row-Level Sampling Operators

Evaluating a query on a probabilistic table (or tables) produces as output another probabilistic table. Though the raw probabilistic data has value, the ultimate goal is to compute statistical metrics: expectations, stddev, etc. To achieve this goal, PIP provides a set of sampling operators: functions that convert probabilistic data into deterministic values.

[2] PIP uses a fixed size burn-in period. Further extensions to PIP's metropolis implementation are possible, but beyond the scope of this dissertation.

Example 3.3.2 *Consider the c-tables*

R	A	ϕ_1
	5	$(Y_1 > 4)$

S	B	ϕ_2
	X	$(Y_2 > 2)$

The query

```
SELECT A * B AS C FROM R, S;
```

produces the result table

T	C	ϕ
	$5 \cdot X$	$(Y_1 > 4) \wedge (Y_2 > 2)$

While debugging a query an end-user might find these query results (that is, results expressed as a lineage) useful. However, it is of limited use for an end-user to know that T contains one row with C equal to $5 \cdot Y_1$ in worlds described by variables $Y_1 > 4$ and $Y_2 > 2$, and is empty in all other worlds – this information essentially restates the query. Analyzing large volumes of this data requires histograms or statistical metrics of aggregates (i.e., expectations of sums, etc...).

Sampling operators accept an expression to be evaluated and the c-table condition (as a conjunctive boolean formula of constraints), henceforth referred to as the row's context. The sampling operator's output is a statistical metric (expectation, standard deviation) or histogram of the expression's value given the context.

PIP focuses on sampling operators that follow *per-row sampling semantics*. Under these semantics, each row is sampled independently. The metric being computed is computed only over the volume of probability space defined by

the expression's context for each row. For example, in the case of Monte-Carlo sampling, samples are generated for each row, but only samples satisfying the row's context are considered. All other samples are discarded. If the context is unsatisfiable, a value of NAN will result.

The choice to focus on per-row sampling operators is motivated by efficiency concerns. If the results table is larger than main memory, the sampling process can become IO-bound. Per-row sampling operators require only a single pass (or potentially a second pass if additional precision is required) over the results. While we do consider table-wide sampling semantics out of necessity for some aggregates, the development of additional table-wide techniques is beyond the scope of this dissertation.

Note that the resulting aggregates are still probabilistic data. For example, the expectation of a given cell is computed in the context of the cell's row; The expectation is computed only over those worlds where the row's condition holds, as in all other worlds the row does not exist. In order to compute the probability of satisfying the row's condition, also referred to as the row's confidence, we define a confidence operator. If the confidence operator is present, all conditions applying to the row are removed from the result and the resulting table is deterministic.

PIP's sampling process for computing expectations, including all techniques described in Section 3.3.1 is summarized in Algorithm 3. Despite the limited number of sampling techniques employed, this algorithm demonstrates the breadth of state available to the PIP framework at sample-time. Independent group sampling requires set of constraints. CDF sampling requires distribution-specific knowledge. Metropolis sampling requires similar knowledge, and also employs

bounds on $P[reject]$ to make efficiency decisions. All of this information is available to the expectation operator, making it the ideal place to implement these, as well as more advanced optimization techniques.

Algorithm 3: `expectation`(E,C,$getP$,(ϵ, δ))

Require: An expression E, a c-table condition C, and confidence bounds (ϵ, δ).

Ensure: The expectation of expression E, given that the condition C holds; if $getP$ is true, also compute the probability that condition C holds.

1: $target \leftarrow \sqrt{2}\cdot\text{erf}^{-1}(1-\epsilon)$; $N \leftarrow 0$; $sum \leftarrow 0$; $sumsq \leftarrow 0$

2: **for all** Variable Groups $K \in C$ s.t. $\exists X \in K$ **and** $X \in E$ **do**

3: $Count[K] \leftarrow 0$; **forall** $X \in K : Sampler[X] \leftarrow$ `Natural` **end for**

4: $S \leftarrow$ `consistencyCheck`(K) {A bounds map (S) is generated as a side effect}

5: **if** `consistencyCheck` failed **then** `ERROR` **end if**

6: **for all** bounded X with available $\text{CDF}^{\pm 1}$ **do** $Sampler[X] =$ `CDF` **end for**

7: **while** $\left(target \cdot \left|(\frac{sum}{N})^2 - \frac{sumsq}{N}\right| + \frac{sum}{N}\right) < (\delta \cdot sum)$ **and** $N < 1/delta$ **do**

8: $N = N + 1$; **for all** K **do** $sample_K \leftarrow \{\}$ **end for**

9: **for all** Variable Groups $K \in C$ s.t. $\exists X \in K$ **and** $X \in E$ **do**

10: **if** $\frac{(Count[K]-N)}{Count[K]} > threshold_{met}$ **then** $Sampler[X \in K] \leftarrow$ `Metropolis` **end if**

11: **if** $Sampler[X \in K] =$ `Metropolis` **then** $sample_K \leftarrow$ `metropolis`(K)

12: **else repeat** $sample_K \leftarrow \{x | x = Sampler[X \in K]()\}$; $Count[K]$++

13: **until** $sample_K$ satisfies E's conditions **end if**

14: $sum \leftarrow sum + E(\cap_K\{sample_K\})$; $sumsq \leftarrow sumsq + [E(\cap_K\{sample_K\})]^2$

15: $Prob \leftarrow \prod_{K \text{ without } \texttt{Metropolis}} \frac{N}{Count[K]} \cdot \prod_{X \in K : Sampler[X] = \texttt{CDF}} \texttt{CDF}(X.high) - \texttt{CDF}(X.low)$

16: **if** $getP$ **then** $Prob \leftarrow Prob \cdot \prod_{K \text{ with } \texttt{Metropolis}} \texttt{probability}(K, E)$ **end if**

17: **return** $(\frac{sum}{N}, Prob)$

Algorithm 4: `probability`(K, E)

Require: A variable group K and an expression E

Ensure: The probability *Prob* of a random sample from the variables in K satisfying E.

1: **if** K has only one variable, and $X \in K$ has an available CDF **then**
2: **return** $\sum_{region \in \text{bounds}(X)} \texttt{CDF}(region.high) - \texttt{CDF}(region.low)$
3: **else**
4: $total \leftarrow 0$; $hits \leftarrow 1$
5: **repeat**
6: $total$++
7: $sample \leftarrow \{x | x = Sample(X)\}$
8: **if** *sample* satisfies E's conditions **then**
9: $hits$++
10: **until** $hits > threshold$
11: **return** $\frac{hits}{total}$

3.3.3 Aggregate Sampling Operators

Aggregate operators (eg. sum, avg, stddev) applied to c-tables introduce a new form of complexity into the sampling process: the result of an aggregate operator applied to a c-table is difficult to represent and sample from. Even if the values being aggregated is a constant, each row's context must be evaluated independently. The result is 2^n possible outputs, each with a linear number of conditions in the number of rows. If the values being aggregated are variable expressions, the result is an identical number of outputs, each containing data linear in the size of the table.

Fortunately, such operators typically appear at the root of a query plan, making them an ideal point at which to perform sampling. Aggregates compute expectations over entire tables, so the probability of a given row's presence in the table can be included into the aggregate's expectation computation.

We begin with the simplest form of aggregate expectation, that of an aggregate that obeys linearity of expectation ($E[f(\vec{y})] = f(E[\vec{y}])$), such as sum(). Such aggregates are straightforward to implement: per-row expectations of $f(\vec{y})\chi(\vec{y})$ are computed, and aggregated (e.g., summed up). Of note however, is the effect that the operator has on the variance of the result. In the case of sum(), each expectation can be viewed as a normally distributed random variable with a shared, predetermined variance. By the law of large numbers, the sum of a set of N random variables with equal standard deviation σ has a variance of $\frac{\sigma}{\sqrt{N}}$. In other words, when computing the expected sum of N variables, we can reduce the number of samples taken for each individual element by a factor of $\frac{1}{\sqrt{N}}$.

If the operator does not obey linearity of expectation (e.g., the max aggregate), the aggregate implementation is more difficult. Any aggregate may still be implemented naively by evaluating it in parallel on a set of sample worlds instantiated prior to evaluation. This is a worst-case approach to the problem; it may be necessary to perform a second pass over the results if an insufficient number of sample worlds are generated. However, more efficient special case aggregates, specifically designed to compute expectations are possible.

Example 3.3.3 *Consider the max() aggregate. If the target expression is a constant, this aggregate can be implemented extremely efficiently. Given a table sorted by the target expression in descending order, PIP estimates the probability that the first element in the table (the highest value) is present. The aggregate expectation is initialized as the*

product of this probability and the first element. The second term is maximal only if the first term is not present; when computing the probability of the second term, we must compute the probability of all the second term's constraint atoms being fulfilled while at least one of the first atom's terms is not fulfilled. Though the complexity of this process is exponential in the number of rows, the probability of each successive row being maximal drops exponentially.

To illustrate this, consider the table (annotated with probabilities)

R	A	ϕ	$P[\phi]$
	5	$X \geq 7$	*0.7*
	4	$Y \geq 7$	*0.8*
	1	$Z \geq 7$	*0.3*
	0	$Q \geq 7$	*0.6*

Based on the probabilities listed above,

$$E[\max(A)] = 5 \cdot 0.7 + 4 \cdot 0.8 + 1 \cdot 0.3 + 0 \cdot 0.6$$

However, if the desired precision is 0.1*, we can stop scanning after the second record since the maximum any later record can change the result is* $1 - (1 - 0.7) * (1 - 0.8) = 0.056$.

3.4 Implementation

In order to evaluate the viability of PIP's c-tables approach to continuous variables, we have implemented an initial version of PIP as an extension to the PostgreSQL DBMS. PIP's extended functionality is provided by a set of user-defined functions written in C, and is illustrated in Figure 3.3.

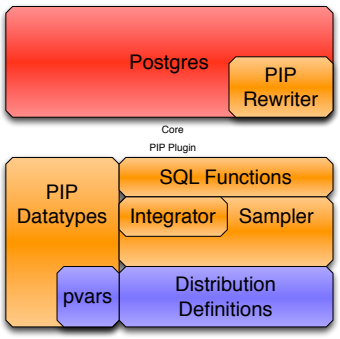


Figure 3.3: The PIP Postgres plugin architecture

3.4.1 Query Rewriting

Much of this added functionality takes advantage of PostgreSQL's extensibility features, and can be used "out-of-the-box". For example, we define the function

`CREATE_VARIABLE`(*distribution*[,*parameters*])

which is used to create continuous variables[3]. Each call allocates a new variable, or a set of jointly distributed variables and initializes it with the specified parameters. When defining selection targets, operator overloading is used to make random variables appear as normal variables; arbitrary equations may be constructed in this way.

Example 3.4.1 *Continuous variables may be created inline with the create-variable operation.*

```
SELECT o.order_id, o.item_id,
       CREATE_VARIABLE ('Normal', p.mean, p.std_dev)
```

[3]For discrete distributions, PIP uses the core functionality of [54], including its `repair-key` operator

```
          AS delivery_time
  FROM orders o, params p
 WHERE o.item_id = p.item_id;
```

Angle brackets around a random variable are shorthand for the variable's expectation. All instances of this are replaced by a call to PIP's expectation sampling function.

To complete the illusion of working with static data, we have modified PostgreSQL itself to add support for c-table constructs. Under the modified PostgreSQL when defining a datatype, it is possible to declare it as a CTYPE; doing so has the following three effects:

- CTYPE columns (and conjunctions of CTYPE columns) may appear in the WHERE and HAVING clauses of a SELECT statement. When found, the CTYPE components of clause are moved to the SELECT's target clause.

  ```
  SELECT * FROM inputs WHERE X>Y and Z like '%foo'
  ```

  is rewritten to

  ```
  SELECT *, X>Y FROM inputs WHERE Z like '%foo'
  ```

- SELECT target clauses are rewritten to ensure that all CTYPE columns in input tables are passed through.

  ```
  SELECT X,Y FROM inputs
  ```

  is rewritten to

  ```
  SELECT X,Y,inputs.phi1,inputs.phi2,... FROM inputs
  ```

There is one exception to this rule – certain functions can be used to make the data deterministic (e.g., by computing and outputting the confidence of a particular row and eliminating all columns with uncertain data, or an aggregate function which performs per-table sampling). Such functions are explicitly labeled as such when they are first defined, and cause the CTYPE rewriter to not pass-through CTYPE columns.

- In the case of aggregates, the mechanism by which CTYPE columns may be passed through is unclear. Thus if the select statement contains an aggregate and one or more input tables have CTYPE columns, the query causes an error unless the aggregate is labeled as a probability-removing function.
- UNION operations are rewritten to ensure that the number of CTYPE columns in their inputs is consistent. If one input table has more CTYPE columns of a given type than the other, the latter is padded with NULL constraints.

  ```
  SELECT * FROM left(X,phi1) UNION right(X,phi2,phi3)
  ```

  is rewritten to

  ```
  SELECT * FROM (SELECT *,NULL FROM left) UNION right
  ```

Note that these extensions are not required to access PIP's core functionality; they exist to allow users to seamlessly use deterministic queries on probabilistic data as illustrated in Figure 3.4.

PIP takes advantage of this by encoding constraint atoms in a CTYPE datatype; Overloaded > and < operators return a constraint atom instead of a boolean if

R_{ctable}	A	B	ϕ
	$X * 3$	5	$X > Y \wedge Y > 3$
	Y	3	$Y < 3 \vee X < Y$

⇓⇓⇓

R_{int}	A (VarExp)	B (integer)	ϕ_1 (CTYPE)	ϕ_2 (CTYPE)
	$X * 3$	5	$X > Y$	$Y > 3$
	Y	3	$Y < 3$	NULL
	Y	3	$X < Y$	NULL

Figure 3.4: Internal representation of C-Tables

a random variable is involved in the inequality, and the user can ignore the distinction between random variable and constant value (until the final statistical analysis).

3.4.2 Defining Distributions

PIP's primary benefit over other c-tables implementations is its ability to admit variables chosen from arbitrary continuous distributions. These distributions are specified in terms of general distribution classes, a set of C functions that describes the distribution. In addition to a small number of functions used to parse and encode parameter strings, each PIP distribution class defines one or more of the following functions.

- `Generate(Parameters, Seed)` uses a pseudorandom number generator to generate a value sampled from the distribution. The seed value allows PIP to limit the amount of state it needs to maintain; multiple calls to Generate with the same seed value produce the same sample, so only the seed value need be stored.
- `PDF(Parameters, x)` evaluates the probability density function of the

distribution at the specified point.

- `CDF(Parameters, x)` evaluates the cumulative distribution function at the specified point.
- `InverseCDF(Parameters, Value)` evaluates the inverse of the cumulative distribution function at the specified point.

PIP requires that all distribution classes define a Generate function. All other functions are optional, but can be used to improve PIP's performance if provided; The supplemental functions need only be included when known methods exist for evaluating them efficiently.

Future implementations could conceivably generalize the sampling process. A sample may be generated using any of the four functions: The Metropolis-Hastings algorithm can sample from an arbitrary PDF, the inverse CDF evaluated on a uniform random value produces a sample, and a binary search may be used to evaluate the inverse CDF given the CDF.

3.4.3 Sampling Functionality

PIP provides several functions for analyzing the uncertainty encoded in a c-table. The two core analysis functions are conf() and expectation().

- `conf()` performs a conjunctive integration to estimate the probability of a specific row's condition being true. For tables of purely conjunctive conditions, conf() can be used to compute each row's confidence.

- `aconf()`, a variant of conf(), is used to perform general integration. This function is an aggregate that computes the joint probability of all equivalent rows in the table, a necessity if disjunctions are in use.
- `expectation()` computes the expectation of a variable by repeated sampling. If a row is specified when the function is called, the sampling process is constrained by the constraint atoms present in the row.
- `expected_sum()`, `expected_max()` are aggregate variants of expectation. As with expectation() they can be parametrized by a row to specify constraints.
- `expected_sum_hist()`,`expected_max_hist()` are similar to the above aggregates in that they perform sampling. However, instead of outputting the average of the results, it instead outputs an array of all the generated samples. This array may be used to generate histograms and similar visualizations.

Aggregates pose a challenge for the query phase of the PIP evaluation process. Though it is theoretically possible to create composite variables that represent aggregates of their inputs, in practice it is infeasible to do so. The size of such a composite is not just unbounded, but linear in the size of the input table. A variable symbolically representing an aggregate's output could easily grow to an unmanageable level. Instead, PIP limits random variable aggregation to the sampling phase.

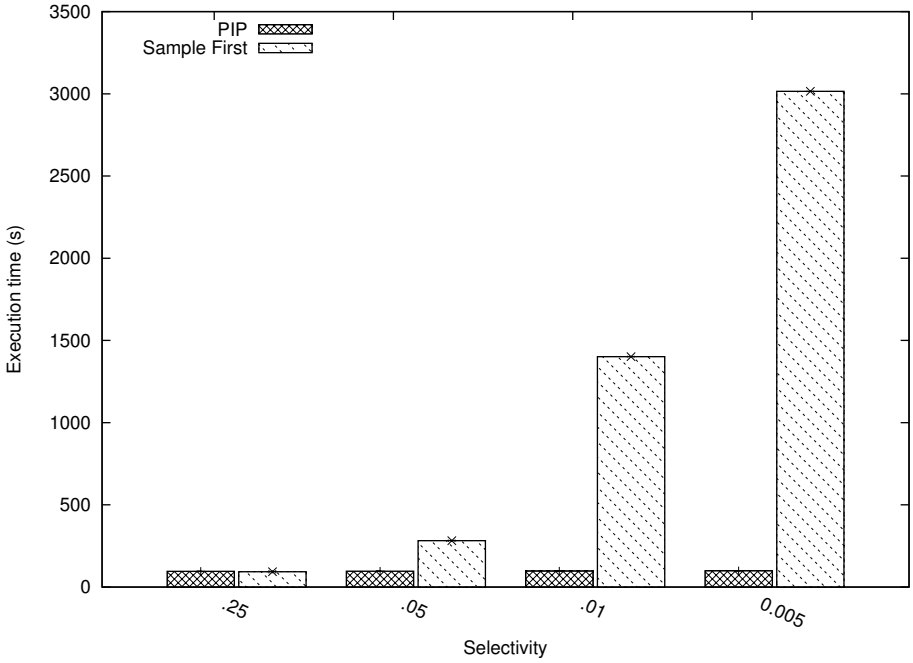


Figure 3.5: Time to complete a 1000 sample query, accounting for selectivity-induced loss of accuracy.

3.5 Evaluation

A sample-first probabilistic extension to Postgres has been constructed as a comparison point for PIP's ability to manage continuous random variables. This extension emulates MCDB's tuple-bundle concept using Postgres user-defined datatypes rows. A sampled variable is represented using an array of floats, while the tuple bundle's presence in each sampled world is represented using a densely packed array of booleans. In lieu of an optimizer, test queries were constructed by hand so as to minimize the lifespan of either array type.

Using Postgres as a basis for both implementations places them on an equal footing with respect to DBMS optimizations unrelated to probabilistic data. This makes it possible to focus the comparison solely on new, probabilistic function-

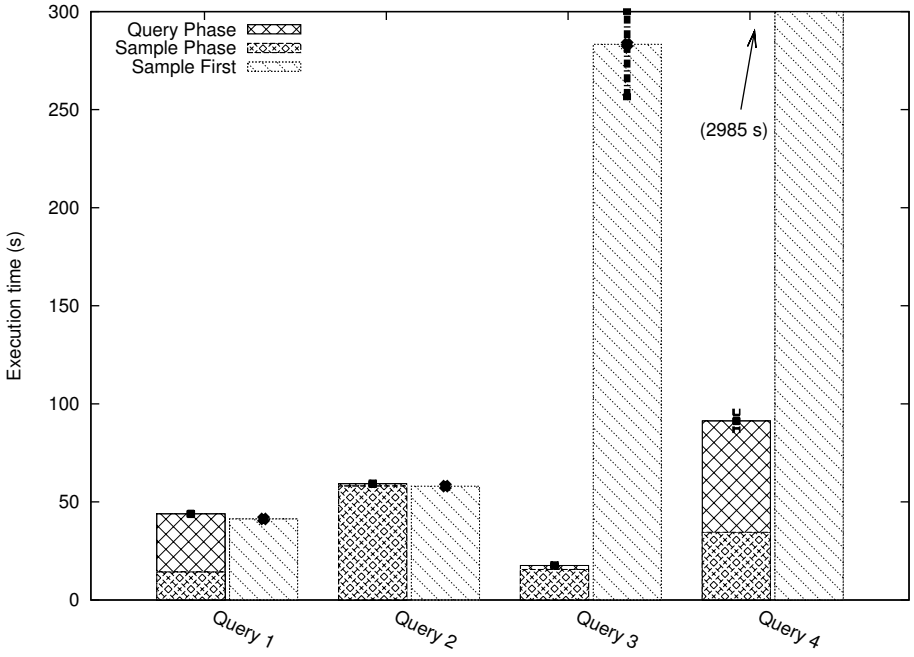
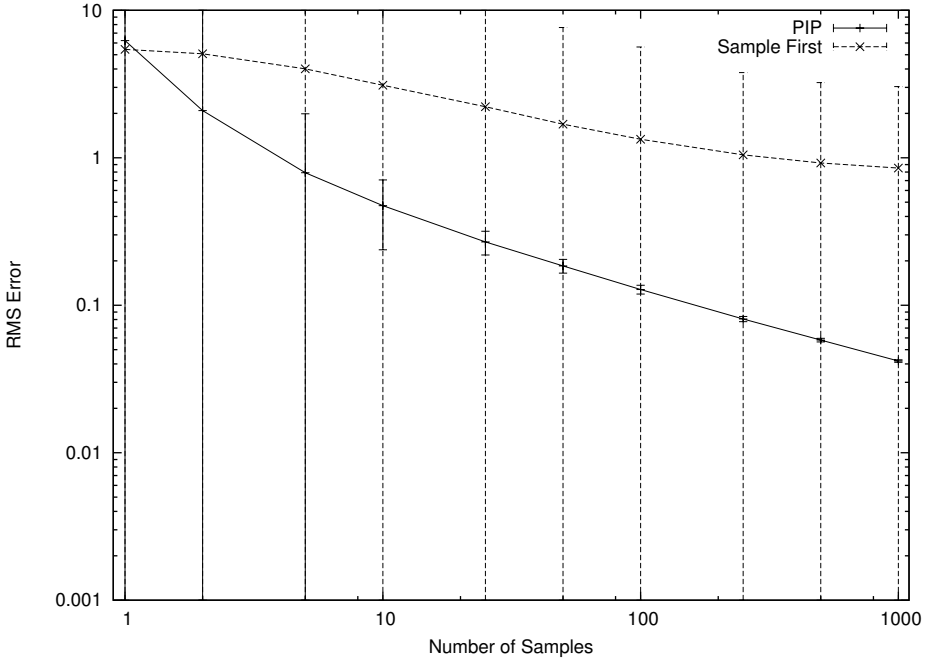


Figure 3.6: Query evaluation times in PIP and Sample-First for a range of queries. Sample-First's sample-count has been adjusted to match PIP's accuracy.

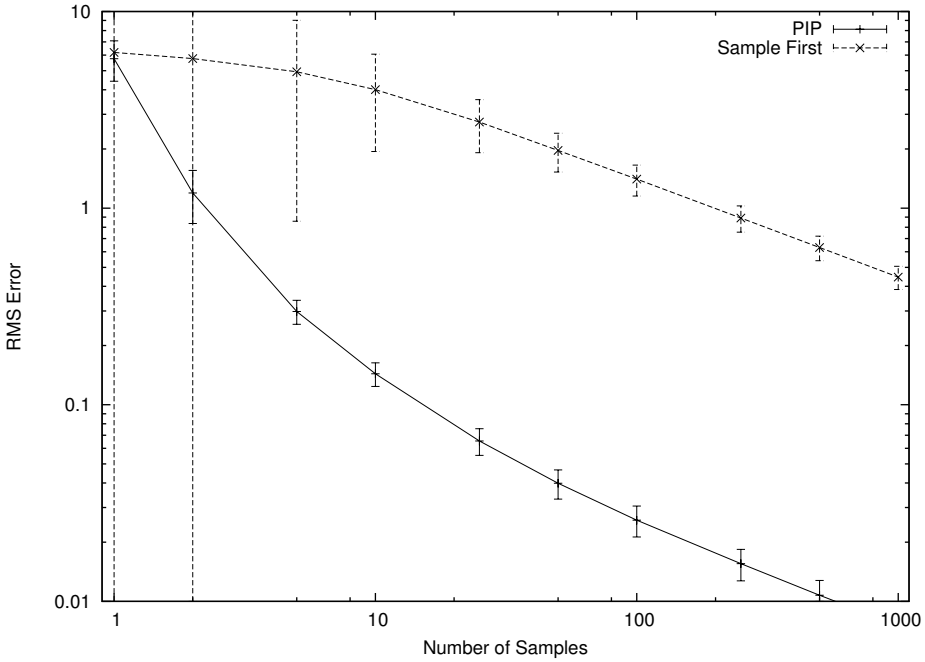
ality added by either system. However, to make the distinction from MCDB (which is a separate development not based on Postgres) explicit, the Postgres extension is referred to as Sample-First.

Both the PIP C-Tables and the Sample-First infrastructures were evaluated against a variety of related queries. Tests were run over a single connection to a modified instance of PostgreSQL 8.3.4 with default settings running on a 2x4 core 2.0 GHz Intel Xeon with a 4MB cache. Unless otherwise specified, queries were evaluated over a 1 GB database generated by the TPC-H benchmark, all sampling processes generate 1000 samples apiece, and results shown are the average of 10 sequential trials with error bars indicating one standard deviation.

First, PIP's performance is demonstrated on a simple set of queries ideally



(a)



(b)

Figure 3.7: RMS error across the results of 30 trials of (a) a simple group-by query Q_4 with a selectivity of 0.005, and (b) a complex selection query Q_5 with an average selectivity of 0.05.

suited to the strengths of Sample-First. These two queries (identical to Q1 and Q2 from [46]) involve parametrizing a table of random values, applying a simple set of math operations to the values, and finally estimating the sum of a large aggregate over the table.

The first query computes the rate at which customer purchases have increased over the past two years. The percent increase parametrizes a Poisson distribution that is used to predict how much more each customer will purchase in the coming year. Given this predicted increase in purchasing, the query estimates the company's increased revenue for the coming year.

In the second query, past orders are used to compute the mean and standard deviation of manufacturing and shipping times. These values parametrize a pair of Normal distributions that combine to predict delivery dates for each part ordered today from a Japanese supplier. Finally, the query computes the maximum of these dates, providing the customer with an estimate of how long it will take to have all of their parts delivered.

The results of these tests are shown as query Q_1 and Q_2, respectively, in Figure 3.6. Note that performance times for PIP are divided into two components: query and sample, to distinguish between time spent evaluating the deterministic components of a query and building the result c-table, and time spent computing expectations and confidences of the results. The results are positive; the overhead of the added infrastructure is minimal, even on queries where Sample-First is sufficient. Furthermore, especially in Q_2, the sampling process comprises a relatively small portion of the query; additional samples can be generated without incurring the nearly 1 minute query time.

The third query Q_3 in Figure 3.6 combines a simplified form of queries Q_1 and Q_2. Rather than aggregating, the query compares the delivery times of Q_2 to a set of "satisfaction thresholds." This comparison results in a (probabilistic) table of dissatisfied customers that is used in conjunction with Q_1's profit expectations to estimate profit lost to dissatisfied customers. A query of this form might be run on a regular basis, perhaps even daily. As per this usage pattern, the component of this query unlikely to change on a daily basis: the expected shipping time parameters were pre-materialized. Q_3 is described in more detail in the Appendix Section A.

Though PIP and Sample-First both take the same amount of time to generate 1000 samples under this query, the query's selectivity causes Sample-First to disregard a significant fraction of the samples generated; Because Sample-First instantiates samples at the granularity of an entire, a sample where the selectivity predicate (based on shipping time) is false does not contribute to the expectation of the profit lost. Thus, for the same amount of work, Sample-First generates a less accurate answer.

To illustrate this point further, see Figure 3.7(a). This figure shows the root-mean-squared (RMS) error, normalized by the correct value in the results of a query for predicted sales of 5000 parts in the database, given a Poisson distribution for the increase in sales and a popularity multiplier chosen from an exponential distribution. As an additional restriction, the query considers only the extreme scenario where the given product has become extremely popular (resulting in a selectivity of $e^{-5.29} \approx 0.005$).

RMS error was computed over 30 trials using the algebraically computed correct value as a mean, and then averaged over all 5000 parts. Note that PIP's

error is over two orders of magnitude lower than the sample-first approach for a comparable number of samples. This corresponds to the selectivity of the query; as the query becomes more selective, the sample-first error increases. Furthermore, because CDF sampling is used to restrict the sampling bounds, the time taken by both approaches to compute the same number of samples is equivalent.

A similar example is shown in Figure 3.7(b). Here, a model is constructed for how much product suppliers are likely to be able to produce in the coming year based on an Exponential distribution, and for how much product the company expects to sell in the coming year as in Q_1. From this model, the expected underproduction is computed, with a lower bound of 0; the selection criterion considers only those worlds where demand exceeds supply. For the purposes of this test, the model was chosen to generate an average selectivity of 0.05. Though the comparison of 2 random variables necessitates the use of rejection sampling and increases the time PIP spends generating samples, the decision to drop a sample is made immediately after generating it; PIP can continue generating samples until it has a sufficient number, while the Sample-First approach must rerun the entire query.

Note the relatively large variance in the RMS error of the Sample-First results these figures, particularly the first one. Here, both the selectivity and the price for each part vary with the part. Thus, some parts become more important while others become harder to sample from. In order to get a consistent answer for the entire query Sample-First must provision enough samples for the worst case, while PIP can dynamically scale the number of samples required for each term.

Returning to Figure 3.6, Queries Q_3 and Q_4 have been run with PIP at a fixed 1000 samples. As Sample-First drops all but a relatively small number of samples corresponding to the selectivity of the query, Sample-First was run with a correspondingly larger number of samples. For Query Q_3, the average selectivity of 0.1 resulted in Sample-First discarding 90% of its samples. To maintain comparable accuracies, Sample-First was run at 10,000 samples.

Figure 3.5 expands on this datapoint, showing the results of evaluating Q_4, altered to have varying selectivities. The sample-first tests are run with $\frac{1}{selectivity}$ times as many samples as PIP to compensate for the lower error, in accordance with Figure 3.7(a). Note that selectivity is a factor that a user must be aware of when constructing a query with sample-first while PIP is able to account for selectivity automatically, even if rejection sampling is required.

It should also be noted that both of these queries include two distinct, independent variables involved in the expectation computation. A studious user may note this fact and hand optimize the query to compute these values independently. However, without this optimization, a sample-first approach will generate one pair of values for each customer for each world. As shown in the RMS error example, an arbitrarily large number of customer revenue values will be discarded and the query result will suffer. In this test, customer satisfaction thresholds were set such that an average of 10% of customers were dissatisfied. Consequently sample-first discarded an average of 10% of its values. To maintain comparable accuracies, the sample-first query was evaluated with 10,000 samples while the PIP query remained at 1000 samples.

As a final test, PIP and the Sample-First implementation were evaluated on the NSIDC's Iceberg Sighting Database[81] for the past 4 years. 100 virtual ships

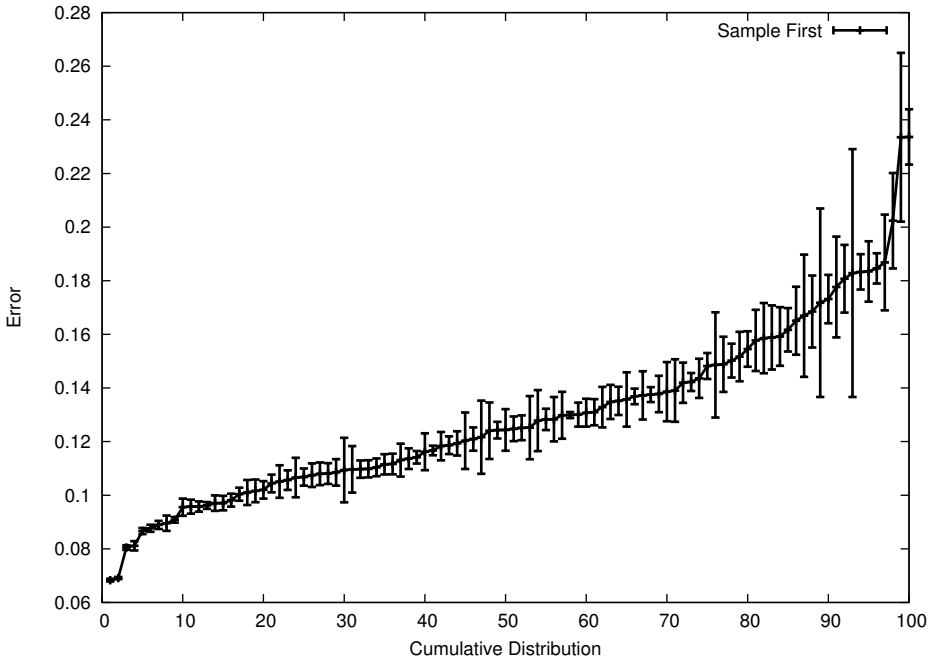


Figure 3.8: Sample-First error as a fraction of the correct result in a danger-estimation query on the NSIDC's Iceberg Sighting Database. PIP was able to obtain an exact result.

were placed at random locations in the North Atlantic, and each ship's location was evaluated for its proximity to potential threats; Each iceberg in the database was assigned a normally distributed position relative to its last sighting, and an exponentially decaying danger level based on time since last sighting. Recently sighted icebergs constituted a high threat, while historic sightings represented potential new iceberg locations. The query identified icebergs with greater than a 0.1% chance of being located near the ship and estimated the total threat posed by all potentially nearby icebergs. The results of this experiment are shown in Figure 3.5. PIP was able to employ CDF sampling and obtain an exact result within 10 seconds. By comparison, the Sample-First implementation generating 10,000 samples took over 10 minutes and produced results deviating by as much as 25% from the correct result on a typical run.

CHAPTER 4

JIGSAW

This chapter describes Jigsaw, a tool for evaluating and optimizing parameterized what-if scenarios in the presence of uncertain data. This sort of parameter optimization and exploration, especially over predictive models is a key component of effective resource allocation in large corporations, and in many other scenarios. Yet, prior to the publication of Jigsaw, no other system had explored, or developed efficient techniques to address this extremely important application of probabilistic databases.

Jigsaw was developed during my internship at Microsoft Research, in collaboration with Suman Nath from Microsoft Research, as well as Charles Loboz, Slawek Smyl, and Steve Lee from Microsoft Corporation. Copyrights on Jigsaw are owned by Microsoft. Jigsaw was originally published at SIGMOD 2011 [52, 51].

4.1 Simulating Business Scenarios

Batch mode execution. Recall the Enterprise Cluster Provisioning example from Section 1.1. An analyst wishes to use models for CPU core demands and availability to determine the optimal date and volume for several server purchase orders to keep the risk of running out of available CPU cores below a certain threshold. The later the purchases occur, the lower the hardware's upkeep costs, but the greater the chance that cores will be unavailable when needed. The question of an ideal purchase date and volume is a simple constrained optimization problem.

A Jigsaw user would specify this optimization problem in three stages: (1) The user defines stochastic models forecasting CPU core availability and demand, (2) The user specifies inter-model interactions to describe the scenario,

and (3) Jigsaw solves the optimization problem by exploring the parameter space of purchase dates and volumes.

For step (1), the user defines various stochastic models that Jigsaw uses as black boxes. These stochastic black boxes are essentially functions that produce samples[1] drawn from the probability distribution that they intend to describe. This framework allows analysts to easily import externally defined models that describe a wide variety of processes and system characteristics. In this specific example, the user writes the following two functions (e.g. based on a model derived in a statistical modeling application like R):

```
DemandModel(current_week, feature_release);
CapacityModel(current_week, purchase1, purchase2);
```

The `DemandModel` function produces a stochastic CPU core usage demand forecast for a given week in the future, taking into account expected future user arrival rates, individual user capacity requirements, and expected user reactions to planned special offers and system features.

The `CapacityModel` function outputs a stochastic estimation of the number of CPU cores available on a given date in the future (given a set of future purchase dates). It also takes into account the current CPU core availability, future expected failure rates, and prediction, based on prior purchasing experiences of when new cores will come online after purchase.

For steps (2) and (3), the user writes the SQL-like query in Figure 4.1. The

[1]Canonical VG-Functions in MCDB produce tables as output. For clarity, I use a simplified notion of stochastic black-box functions that produce only single values. To make this distinction explicit, the term black-box function is used. Naive extensions of Jigsaw's fingerprinting technique to full VG-functions are trivial (e.g., extend the function with row and column id parameters) and optimized extensions are relegated to future work.

```
-- DEFINITION --
DECLARE PARAMETER @current_week AS RANGE 0 TO 52 STEP BY 1;
DECLARE PARAMETER @purchase1 AS RANGE 0 TO 52 STEP BY 4;
DECLARE PARAMETER @purchase2 AS RANGE 0 TO 52 STEP BY 4;
DECLARE PARAMETER @feature_release AS SET (12,36,44);

SELECT DemandModel(@current_week, @feature_release)
          AS demand,
       CapacityModel(@current_week, @purchase1, @purchase2)
          AS capacity,
       CASE WHEN capacity < demand THEN 1 ELSE 0 END
          AS overload
  INTO results;
-- BATCH MODE --
OPTIMIZE SELECT @feature_release, @purchase1, @purchase2
         FROM results
         WHERE MAX(EXPECT overload) < 0.01
         GROUP BY feature_release, purchase1, purchase2
     FOR MAX @purchase1, MAX @purchase2
```

Figure 4.1: An example Jigsaw query.

core of the scenario is a simple SQL `SELECT` query that produces an output `result` table – in this example, containing `capacity`, `demand`, and `overload` columns. Note that, as Jigsaw is built around a probabilistic database (PDB) system, this results table is specified as a probability distribution over the space of possible results. Two aspects of the query require further discussion: (a) The query contains several parameter variables, each prefixed with a `@`. Parameter variables, with their bounds and sets of permitted and initial values, are declared as part of the scenario using `DECLARE PARAMETER` statements and are equivalent to standard SQL variables from the user's perspective. (b) The optimization goal is expressed with an `OPTIMIZE` query, which iterates over the parameter space to find the latest `purchase1` and `purchase2` that keep the expected risk of `overload` (a condition defined as a week when `capacity <`

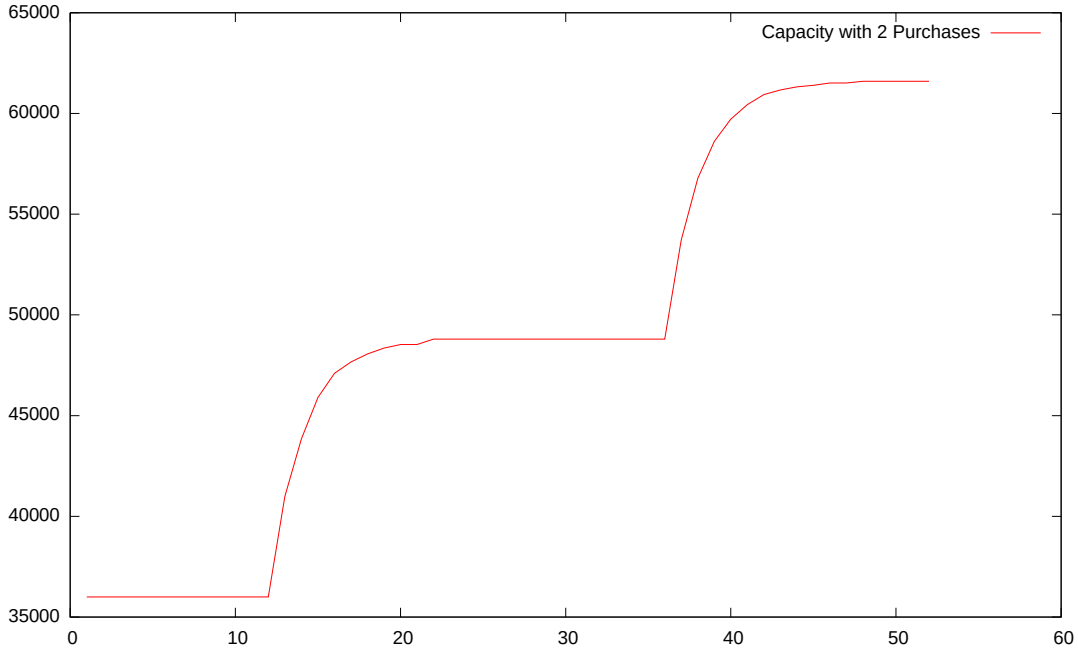


Figure 4.2: Example output of the `CapacityModel` with two purchase dates.

`demand`) within a threshold.

One possible implementation of the `CapacityModel` is of interest. This model is characterized by a sequence of discrete events (e.g., purchases or hardware failures), each affecting the cluster's capacity, as shown in figure 4.1. Each event is produced by a separate model, so the database engine itself can compute the cumulative effect of the events with a simple SQL `SUM` aggregate. Also consider the `CapacityModel` expectation viewed in a time-series plot. Though each purchase has a stable long-term impact on the cluster's capacity, this plot is characterized by two distinct *structures* in the vicinity of each purchase date.

Note that PDB functionality provides a glue layer that allows analysts to define interactions between models. PDBs allow a clean, hierarchical approach to

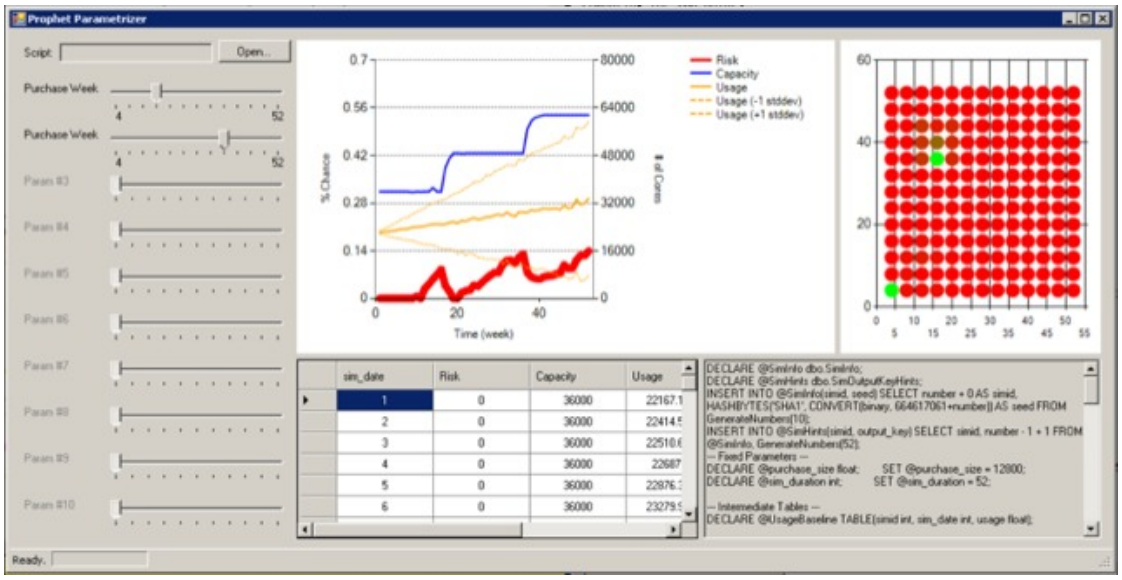


Figure 4.3: Jigsaw's interactive interface.

model construction. For example, forecasting discontinuities identified by expert knowledge are notoriously difficult to incorporate into statistical models. Planned maintenance at a datacenter temporarily reduces the datacenter's capacity by a known amount. By defining `CapacityModel` as a SQL function that applies the effects of planned maintenance to the baseline model, a clean separation exists between the underlying process and the expert knowledge.

Interactive mode. Jigsaw can also be used in an interactive online mode. In this mode, the user modifies various parameter values (e.g., purchase date and volume) and quickly sees the outcome (e.g., the risk of overload at a given date). As parameter values are modified, the system continually updates a progressively refined estimate of the results table for those parameter values. This quickly gives a rough estimate of the final answer so that the user, not finding the given parameter values interesting, can abandon the simulation in the middle and try a different parameter value. This mode is particularly targeted at users who

may not have an extensive statistics background. An analyst-developed scenario can be used by an executive (e.g. as part of a management dashboard tool) to quickly observe the expected outcome of specific financial decisions for various parameter values.

The interactive mode, with the output shown in Figure 4.3, is expressed with the following execution query (parameter definition and `SELECT` portions of the query are same as in Figure 4.1):

```
-- INTERACTIVE MODE --
GRAPH OVER @current_week
      EXPECT overload WITH bold red,
      EXPECT capacity WITH blue y2,
      EXPECT_STDDEV demand WITH orange y2;
```

The query above provides Jigsaw with a parameter to use as the graph's X-Axis, and specifies how each column in the results table is to be graphed in the GUI (Figure 4.3).

4.1.1 Jigsaw Simulation Process

Figure 4.4 shows how a Jigsaw executes an optimization query in the batch mode. Each random table in the uncertain database is represented on disk by its schema, together with a set of black-box functions that are used to generate realizations of uncertain attribute values. When a query is issued, the *Parameter Enumerator* module enumerates all feasible parameter values for the black-box functions involved in the query. This brute force approach is necessary to guar-

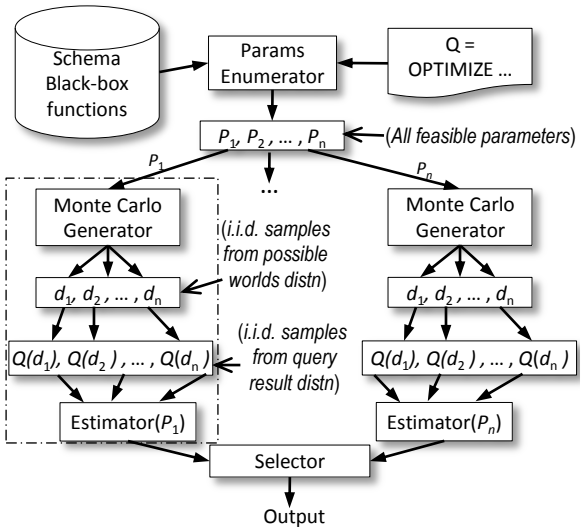


Figure 4.4: Processing optimization queries in Jigsaw

antee that the optimization converges to the global maximum for an arbitrary black-box function. Note that Jigsaw's fingerprinting techniques remain applicable to more advanced techniques that use additional information about the black-box (e.g., gradient-descent, if the black-box is known to be continuous).

For each parameter value, Jigsaw then invokes its PDB subsystem (shown inside the dotted box). The PDB subsystem (loosely modeled after MCDB) invokes the black-box functions with the current set of parameter values to generate a set of $n \geq 1$ independent and identically distributed (i.i.d.) sampled *instances*, sometimes referred to as possible worlds; for parameter valuation P_a, sample d_i is referred to as being generated by instance (P_a, i). Recall that in a PDB, the output of a query is a probability distribution. Evaluating the query over each sampled possible world generates a set of i.i.d. samples of the results table's distribution. These latter samples are then aggregated by the *Estimator* to compute one or more characteristics of interest (i.e., mean, standard deviation, etc.) for the output distribution. The process is repeated for all different param-

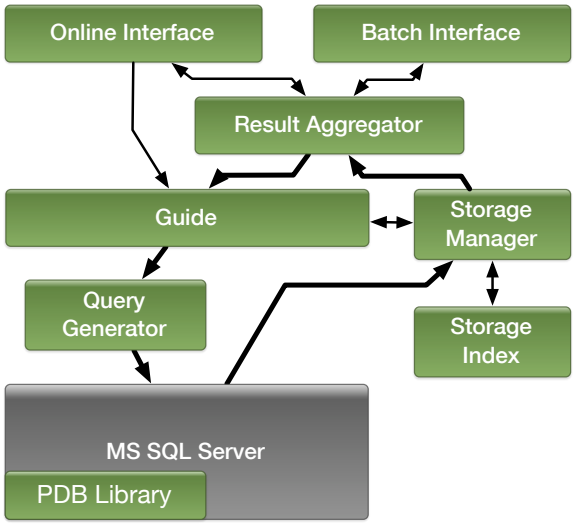


Figure 4.5: Jigsaw's Architecture

eter values. Finally, the *Selector* component selects the parameter value, along with its output distribution, that satisfies the optimization goal.

4.1.2 Jigsaw Architecture

Jigsaw itself is implemented as a wrapper around a commercial database, as shown in Figure 4.5. Similar to MCDB [46], the wrapper performs Monte Carlo style world sampling, evaluates deterministic queries over the samples, and aggregates resulting outputs. The use of tuple-bundles and similar optimizations in MCDB is both orthogonal and complimentary to Jigsaw's primary functionality.

Jigsaw processes a given scenario by iteratively invoking the DBMS' query engine. With each iteration, a component of Jigsaw called the *guide* selects a set of parameters to generate samples for and produces an *instance* table. Each in-

stance is defined by the parameter combination that it represents and an identifier unique among all instances generated for the same parameter combination. Each instance represents a single sample of the query's output, generated for a single parameter combination. Before evaluation, the query is rewritten so that parameter values are drawn from the instance table, and the each row in the query's output is annotated with a pointer to the instance that it belongs to.

After each processing iteration, results are passed to and stored by Jigsaw's storage manager; results are applied to future queries in order to avoid instance evaluation, produce improved estimates, and direct the optimization process.

Finally, an aggregation step is required. Even if the user has not specified an aggregate as part of the batch-mode post-processing query, the output of a query over probabilistic data is itself a distribution over possible results. Metadata provided as part of the `OPTIMIZE` or `GRAPH` statements is used to direct this aggregation process.

4.1.3 Jigsaw Challenges

The most expensive aspect of Jigsaw's simulation process is its interaction with the underlying PDB. This processing overhead is linear in the size of the parameter space and dominates all other processing tasks performed by Jigsaw. The primary goal of Jigsaw is to reduce the number of instances on which the PDB must be invoked. This is achieved by exploiting several observations about redundancy in computations.

First, outputs of many enterprise-related stochastic functions are strongly

correlated under various parameter values (examples in Section 4.2). Identifying such correlations can help to avoid exploring large regions of the parameter space.

Second, in many event-based processes with Markovian dependencies (i.e. each step in the process depends on the output of the prior step), the Markovian dependencies are relevant only in the steps near an event. A suitably crafted non-Markovian estimator function (examples in Section 4.3) may be used to reduce simulation required for the other steps.

Finally, in interactive mode execution, a quick estimation of simulation results for a selected parameter value can often be given based on results from previously selected parameters; the estimation can then be gradually refined with more samples.

To exploit the above observations, Jigsaw needs to address the following challenges.

- How can parameter values that produce the same (or similar) outputs be efficiently identified and exploited?
- How can correlated Markovian steps be efficiently identified and exploited?
- Can an accurate estimate be obtained for one parameter value in interactive mode by reusing results computed for other parameter values?

The remainder of this chapter discusses how Jigsaw addresses these challenges.

4.2 Fingerprints

The key concept Jigsaw uses to reduce the number of Monte Carlo evaluations is *fingerprints*. A fingerprint of a stochastic black box function is a concise and easily-computable data structure that summarizes its output distribution. Thus, a fingerprint can be used to determine a function's *similarity* with another function, or its own instantiations under different parameter values. A concrete example of fingerprints is presented in Section 4.2.1.

The outputs of a deterministic function F evaluated on two different values P_i and P_j, are deemed similar (denoted as $F(P_i) \sim_{\mathcal{M}} F(P_j)$) if there exists a closed form mapping function $\mathcal{M}$ that maps from $F(P_i)$ to $F(P_j)$.

$$F(P_i) \sim_{\mathcal{M}} F(P_j) \equiv F(P_i) = \mathcal{M}(F(P_j))$$

Consider a stochastic function F with output $X = F(P_i)$ and probability distribution $f(x = X|P_i)$. F is similar at P_i and P_j if a closed form mapping function exists to map the domain of $f(x|P_i)$ into that of $f(x|P_j)$.

$$F(P_i) \sim_{\mathcal{M}} F(P_j) \equiv \forall x : f(x|P_i) = f(\mathcal{M}(x)|P_j)$$

More generally, $\mathcal{M}$ can be thought of as the central element of a family of mapping functions that map not only function values but also metrics, aggregates, and other derived values. Efficient translation between members of this family can substantially reduce the sampling requirements of a computation. For example, consider a scenario where both expectations $E[F(P_i)]$ and $E[F(P_j)]$ are needed, and $F(P_i) \sim_{\mathcal{M}} F(P_j)$ can be efficiently proven. An $\mathcal{M}_{expect}$ derived from $\mathcal{M}$ such that $E[F(P_j)] = \mathcal{M}_{expect}(E[F(P_i)])$, eliminates the need to explicitly compute $E[F(P_j)]$.

Identifying the mapping function for an arbitrary pair of stochastic black-box functions $(F(P_i), F(P_j))$ is difficult for two reasons: (1) The functions are black-boxes – interactions with the function are limited to sample generation. (2) The functions are stochastic. In order to match two distributions, it is first necessary to approximate the distributions (i.e., by sampling from both, negating the benefits of having established similarity).

Rather than attempt to map the result distribution, Jigsaw employs a shortcut. The abstract `fingerprint` operation (and corresponding mapping function M_f) efficiently maps parameterized stochastic black-box functions to concise, comparable data structures such that with high probability:

$$F(P_i) \sim_{\mathcal{M}} F(P_j) \equiv \texttt{fingerprint}(F(P_i)) = \mathcal{M}_f(\texttt{fingerprint}(F(P_j)))$$

Fingerprints can be computed for individual stochastic black-box functions, such as `DemandModel` in Figure 4.1, or combinations of such functions. Taken to one extreme, the entire Monte Carlo simulation shown inside the dashed box in Figure 4.4 can be treated as the stochastic function F. Thus, $F(P_i) \sim_{\mathcal{M}} F(P_j)$ implies that expensive Monte Carlo simulations for parameter value P_j can be avoided by accurately estimating the output of $Estimator(P_j)$ as $\mathcal{M}_{est}(Estimator(P_i))$.

4.2.1 Computing Fingerprints

Identifying similarities between the outputs of two functions is hard [58] in general. Jigsaw uses a probabilistic approach based on the principle of *random testing* [38], a well-known software testing technique. For random testing of a deterministic function F against a hypothesis function H, both functions are

evaluated on m random inputs and the results are compared. The function F is declared satisfying the hypothesis H if the outputs of F and H match for all m random inputs. Random testing has two features in particular, that make it well suited to Jigsaw's needs: (1) Random testing is simple and can be used while treating the functions as black boxes. (2) Random testing has been shown to be robust for functions with a small number of conditional branches so that a small number of random inputs can exercise all code paths. In the motivating cloud infrastructure management context that almost all stochastic functions are relatively simple and contain at most one or two conditional branches.[2] Algorithm 4.2.1 shows an example of such a function. This function produces a prediction of weekly usage, which is linearly growing, normally distributed with a discontinuity at the point where *current_week* and $feature$ are equal. The function has only one branching condition.

The same principle determines similarities between the outputs of a stochastic black-box function F under two valuations of the same parameters P_i and P_j. However, unlike random testing where the parameters are random and the function is deterministic, Jigsaw must deal with stochastic functions and fixed parameters. To make F deterministic, F is extended with a seed parameter σ which is used to seed a pseudorandom number generator which replaces all sources of randomness within $F(P_i, \sigma)$. In practice, these modifications are negligible, as randomness is typically obtained from system API calls (e.g. `rand()`).

It is crucial for both invocations of F to use *the same source of randomness* to make their comparison meaningful. Consider two stochastic functions that

[2]The functions are kept simple in practice, modeling only one particular aspect of the system so that they can be trained and validated even with small, noisy data sets.

Algorithm 5: DemandModel(*current_week*, *feature*)

Require: The *current_week* being simulated, and a *feature* release date.

Ensure: The *demand* for the week being simulated.

1: *demand* = `Normal`(
   $\mu : 1 * current\_week$,
   $\sigma^2 : 0.1 * current\_week$
)
2: **if** $current\_week > feature$ **then**
3: *demand* += `Normal`(
   $\mu : 0.2 * (current\_week - feature)$,
   $\sigma^2 : 0.2 * (current\_week - feature)$
)

output 0 and 1 with equal probability. When repeatedly evaluated with the same sequence of random seeds, they can be quickly declared to be equivalent with a very high probability. On the other hand, using different seeds, equivalence testing is much more difficult. Consider the example stochastic function in Algorithm 4.2.1. As a sum of two normal distributions, the function's output is normally distributed for all inputs. Suppose the function is invoked twice as `DemandModel`(1,3) and `DemandModel`(2,4). Both invocations take the same code path, and their outputs will be drawn from linearly correlated distributions. In addition, Using a pseudorandom number generator seeded with the same value for each invocation ensures that there is not just a correlation, but a linear mapping from one fingerprint to the other. In contrast, using different random seeds would hide the one-to-one similarity in their outputs. Figure 4.6 explains this.

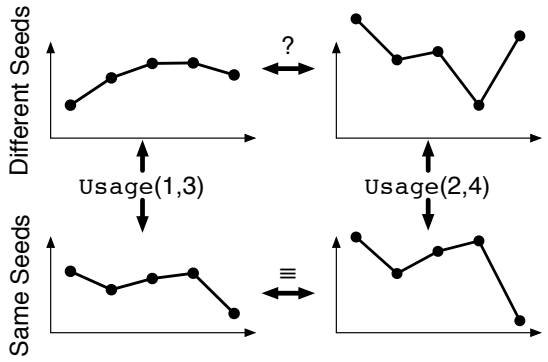


Figure 4.6: The use of identical seed values exposes simple correlations between output distributions for different inputs; When graphing the seed id against function output, such correlations appear as similarly shaped curves.

.

A Concrete Fingerprint. The fingerprint of a parameterized stochastic function $F(P_i)$, with respect to a vector of m seed values $\{\sigma_k\}$, is the vector of size m where the k'th entry of the vector is the output of $F(P_i)$ with σ_k as the random seed. More formally,

$$\texttt{fingerprint}(\{\sigma_k\}, F(P_i)) = \{\theta_k = F(P_i, \sigma_k) | 0 \leq k < m\}$$

Henceforth, this definition of fingerprints will be used implicitly, together with an implicit global seed set $\{\sigma_k\}$, randomly generated as part of Jigsaw's initialization process and held constant throughout.

Note that using the same set of random seeds for different parameter values does not affect the correctness of Jigsaw's Monte Carlo simulations. Referring to Figure 4.4, since the seeds used by each Monte Carlo Generator are i.i.d. random, inputs to the $Estimator(P_i)$ are i.i.d. samples from query result distribution. Thus, the output of $Estimator(P_i)$ remains statistically correct. Using same set of seeds for different parameter values introduces correlated error terms into the

outputs of different Estimators, but the Selector only compares, and never combines, the Estimator's outputs.

Mapping Functions. For fingerprints as defined above, a fingerprint mapping function $\mathcal{M}_f$ can be defined as a simple mapping function $\mathcal{M}$ applied to each element of the fingerprint (in order to identify its similarity with another fingerprint). For example, consider two fingerprints: $\theta_1 = (0, 1.2, 2.3, 1.3, 1.5)$ and $\theta_2 = (0.1, 1.3, 2.4, 1.4, 1.6)$. The mapping function $\mathcal{M}(x) = x + 0.1$ maps θ_1 to θ_2. In general, mapping functions should be: (1) easy to parameterize, (2) easy to validate, (3) easy to compute, (4) easily applied to simple aggregate properties (e.g., expectation).

Given two fingerprints, Jigsaw can automatically compute a linear function (i.e., $\mathcal{M}_{\alpha,\beta}(x) = \alpha x + \beta$) that maps one fingerprint to another, if such a mapping exists (Algorithm 6). Linear mapping functions fulfill the desired characteristics precisely: (1) The mapping function can be determined from two distinct values in a pair of fingerprints. (2) The remaining values in the fingerprints can be used to validate the mapping. (3) Linear functions are incredibly simple, and (4) can be easily applied to simple aggregate properties such as expectation and standard deviation.

In general, the notion of similarity between two signatures is application dependent. Therefore, Jigsaw allows users to provide their own classes of mapping functions.

Using Fingerprints. With fingerprints, Jigsaw executes Monte Carlo simulations for different parameter values as follows. Let F denote the entire Monte Carlo simulation with a parameter value P_i (i.e., the computation inside the

dashed box in Figure 4.4). Thus, the fingerprint of $F(P_i)$ is essentially the outputs of first m simulation rounds with parameter P_i.

During execution, Jigsaw incrementally maintains a set of *basis distributions*. Each basis distribution is a tuple (θ_i, o_i), implying that Jigsaw has already computed the output metrics o_i for some $F(P_i)$ with fingerprint θ_i. For a new parameter value P_j, Jigsaw first computes fingerprint θ_j of $F(P_j)$ (as part of the first m rounds of simulation with parameter P_j). It then checks for a basis distribution with fingerprint θ_k such that $\theta_j \sim_{\mathcal{M}} \theta_k$. If such a basis distribution exists, Jigsaw omits the subsequent rounds of simulation for P_j and returns $\mathcal{M}_{est}(o_k)$ instead.

Retrieving Mapping Functions. When presented with an unknown distribution, Jigsaw compares each new fingerprint against all the fingerprints in the basis distribution, identifying a mapping to one of them if it exists. Algorithm 7 shows the process. Jigsaw first uses a suitable indexing scheme (described next) to prune the search space of candidate basis fingerprints. For each pairing candidate, Jigsaw uses the `FindMapping` function to discover a possible mapping between the two fingerprints. An instance of the `FindMapping` function, the `FindLinearMapping` function shown in Algorithm 6 searches for mappings of the form $\mathcal{M}(x) = \alpha x + \beta$. If a mapping exists between two fingerprints, Jigsaw uses the mapping to reuse work done for the existing basis distribution. If no mappable fingerprint can be found, Jigsaw completes the simulation process and adds the results (i.e., the fingerprint and computed metric(s)) to the set of basis distributions.

Algorithm 6: $FindLinearMapping(\theta_1, \theta_2)$

Require: Two fingerprints θ_1 and θ_2 of size m

Ensure: A linear function $\mathcal{M}(x) = \alpha x + \beta$ such that $\mathcal{M}(\theta_1[i]) = \theta_2[i], \forall i$, and *null* if no such function exists

1: $\alpha \leftarrow (\theta_2[1] - \theta_2[2])/(\theta_1[1] - \theta_1[2])$
2: $\beta \leftarrow \theta_2[1] - \alpha\theta_1[1]$
3: $match \leftarrow$ **true**
4: **for** $i = 3$ to m **do**
5: **if** $\alpha\theta_1[i] + \beta \neq \theta_2[i]$ **then**
6: $match \leftarrow$ **false**
7: **return** $(\mathcal{M}(x) = \alpha x + \beta)$ **if** *match*, *null* otherwise

4.2.2 Indexing Fingerprints

The existence of $\mathcal{M}$ can be computed quickly for any pair of fingerprints. However, the expected number of times this test must be performed grows linearly with the number of basis distributions.

Instead of naively scanning every basis distribution, Jigsaw builds an index over the basis fingerprints. The goal of indexing is to quickly find a set of candidate basis fingerprints that are similar to a given fingerprint (i.e., where a mapping exists). The set of fingerprints returned by the index must contain all similar fingerprints. In addition, it may contain few fingerprints that are not similar to the given fingerprint; these false positives are later discarded in Algorithm 7.

Currently Jigsaw supports the following two indexing strategies that reduce

Algorithm 7: $FindMatch(F, P_a)$

Require: A stochastic black box function F, and a point in its parameter space P_a.

Ensure: The pair $(basis, \mathcal{M})$, where $basis$ is a basis distribution (fingerprint θ, output metrics o), and $\mathcal{M}$ is a mapping function such that $\theta \sim_{\mathcal{M}}$ `fingerprint`$(F(P_a))$

1: $\theta \leftarrow \{F(P_a, \sigma_i) | i \in [0, m)\}$
2: $candidates \leftarrow CandidateFingerprint(basis, \theta)$
3: **for all** $basis \in candidates$ **do**
4: $\quad \mathcal{M} \leftarrow FindMapping(basis, \theta)$
5: $\quad$ **if** $\mathcal{M} \neq null$ **then**
6: $\quad\quad$ **return** $(basis, \mathcal{M})$
7: **return** $\{[(\theta, \texttt{Estimator}(F(P_a))), (\mathcal{M}(x) = x)]\}$

the cost of matching linear mapped fingerprints to a single hash-table lookup with high probability.

Normalization. The first indexing strategy is to translate the fingerprints to a(n arbitrarily chosen) normal form such that that two *similar* fingerprints have the *same* normal form (and hence can be retrieved by a hash lookup). Such normalization requires a class of mapping function that admits a normal form translation. For example, when using a linear mapping function, a fingerprint's normal form can be produced by taking the first two distinct sample values and identifying the linear translation that maps them to 0 and 1 (or, any two predefined constants) respectively. If two fingerprints have a linear mapping, then *all*, not just the first two, entries of their normal forms will be identical.

Sorted SID. Normalization requires that the mapping function admit a normalized representation of a fingerprint. In some cases (e.g., a probabilistic mapping), no such normal form can be computed easily. In such cases, each sample value in the fingerprint is assigned an identifier (e.g., its index position in the fingerprint), using the same identifier ordering across all fingerprints. Jigsaw then sorts the sample values in a fingerprint, and takes the resulting sequence of sample identifiers (or, SIDs) as the hash key in the index. As long as the mapping function is monotonically increasing, the resultant ordering of SIDs will be consistent across all mappable distributions. Even if the mapping function is only monotonic, a similar effect can be achieved by comparing both the SID sequence and its inverse.

4.3 Markovian Jumps

Jigsaw allows users to specify inter-model dependencies. Consider two models where the first model predicts the release date of a particular feature of the cloud service, and the second model predicts demand, given that release date. Frequently, such dependencies are cyclical: the feature release date might be driven by demand. For example, sufficiently high demand might convince management to allocate additional development resources to the feature.

As a consequence of this sort of cyclical dependency, the models and thus the simulation must be evaluated as a Markovian process, where a model is evaluated in discrete steps and its output for any given step is dependent on the prior step's output. The discrete steps are usually small (e.g., a day in the above example) so that outputs of other models affecting the model remain static within

a step. Every step in the process must be simulated, even if the only output of interest is for one specific step (e.g., user demand in two months).

In the space of cloud logistics, models with this sort of cyclical dependency often have an interesting characteristic: the Markovian dependency is present only over certain steps. In the case of the feature release date, as long as the user demand remains strictly (or at least with high probability) below or above the threshold value, the feature release date is unaffected. For these periods, the demand and feature release date model can be treated as non-Markovian, despite its cyclical dependency. Concretely, Markovian dependencies in this sort of model are characterized as (1) infrequent, and (2) often closely correlated (3) discontinuities in (4) an otherwise non-Markovian process. Thus, given the state of the system at the beginning of one of these non-Markovian regions, it is possible to create a non-Markovian *estimator function* for the remainder of the region.

These infrequent Markovian dependencies occur often in event-based simulations. Forcing programmers to identify the ranges within which these dependencies occur is undesirable. Instead, Jigsaw can automatically identify non-Markovian regions in these processes automatically by using fingerprints. Once a non-Markovian region is identified, the estimator function reproduces the state of the Markov process at the end of the region – When evaluating the Markov process, Jigsaw effectively skips over non-Markovian regions.

4.3.1 Fingerprinting Markov Processes

Consider a model F that needs to be evaluated in a sequence (or a chain) of discrete steps. Assuming that Markovian dependencies are infrequent, outputs of F in many successive steps will not be affected by previous steps. To *jump* over such non-Markovian steps and avoid expensive computation, Jigsaw uses a non-Markovian estimator function E (discussed further in Section 4.3.2), which predicts the outputs of F at different steps of the chain *without* considering the outputs (of F or other models) at previous steps. By comparing the fingerprints of E and F, Jigsaw can efficiently identify the regions over which E is a valid approximation.

Recall that each fingerprint of F is a set of its random outputs. Thus, the fingerprint for any step in a Markov process can be used to generate the fingerprint for the next step. Instead of evaluating the full set of n Monte Carlo simulation rounds of the Markov chain, Jigsaw evaluate only a fingerprint-sized ($m < n$) set and compares it to the fingerprint of an estimator function. If a mapping exists between the two, the estimator remains viable.

To compute the value of a Markovian black-box function at a particular step in the chain, Jigsaw does an exponential-skip-length search of the chain until it finds a point where the estimator ceases to be viable (i.e., it fails to provide a mappable fingerprint). From that point, it does a binary search to find the last point in the chain where the estimator is viable, uses the estimator to rebuild the state of the Markov process, generates the next step, and repeats the process. This process is made explicit in Algorithm 8.

Consider the previous example of a cyclically dependent user demand and

Algorithm 8: $MarkovJump(F_{mkv}, initial, target)$

Require: A function, $F_{mkv}(prev\_state) = new\_state$ describing a Markov process and its estimator, respectively. An *initial* state for the functions. A *target* number of steps to return after. A statically defined fingerprint size m.

Ensure: The state of each instance of the Markov process after *target* steps.

1: $state \leftarrow \{initial, initial, \ldots\}$; $\theta_1 \leftarrow state[0 \ldots m]$
2: $s \leftarrow 1$; $F_{est} \leftarrow F_{mkv}(\theta_1)$
3: **loop**
4: **for** $s/2 < i \leq s$ **do**
5: $\theta_i \leftarrow F_{mkv}(\theta_{i-1})$
6: **if** $(s > target) \wedge (F_{est} \sim_{\mathcal{M}} \theta_{target})$ **then**
7: **return** $\mathcal{M}(F_{est}(state))$
8: **if** $F_{est}(s, state[0 \ldots m]) \sim_{\mathcal{M}} \theta_s$ **then**
9: $s \leftarrow s * 2$
10: **else**
11: $(valid, \mathcal{M}) \leftarrow ArgMax_{valid}(\{(valid, \mathcal{M}) | valid \in [\frac{s}{2}, s] \wedge F_{est} \sim_{\mathcal{M}} \theta_{valid}\})$
12: **if** $valid \leq 1$ **then** $state \leftarrow F_{mkv}(state)$; $valid \leftarrow 1$
13: **else** $state \leftarrow \mathcal{M}(F_{est}(state))$
14: $target \leftarrow target - valid$; $s \leftarrow 1$;
15: $\theta_1 \leftarrow state[0 \ldots m]$; $F_{est} \leftarrow F_{mkv}(\theta_1)$

feature release date models. An example execution of the Markov Jump algorithm is illustrated in Figure 4.7. Jigsaw begins with an estimator for the Markov process that assumes the feature has not yet been released (the initial system state). (4.7.a) It iterates over each step of the Markov process, computing only the fingerprint and not the full set of instances being generated. At each step,

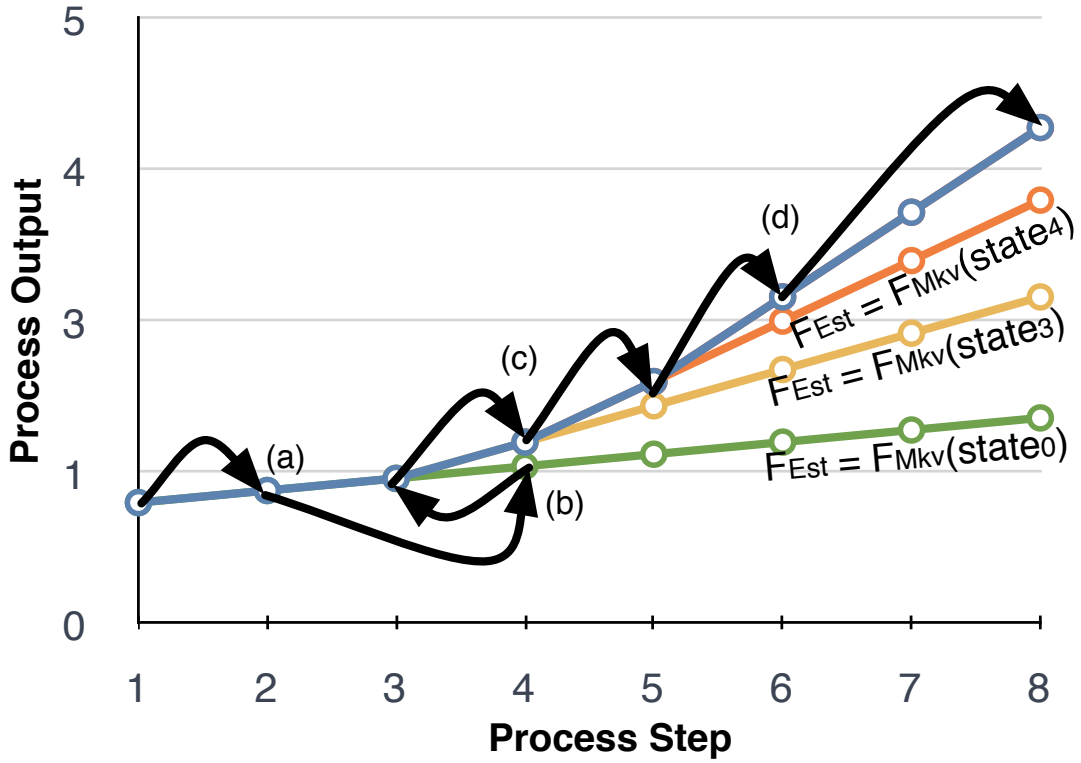


Figure 4.7: An example execution of the Markov Jump algorithm.

the fingerprint of the Markov function is compared to that of the estimator. The number of steps between comparisons grows exponentially until (4.7.b) the algorithm finds a mismatch. (4.7.c) At this point, the algorithm backtracks to the last matching value with a binary search and uses the estimator to regenerate the full state of the Markov process. (4.7.d) The Markov process is used to step the full set of instances until the estimator function once again begins to produce matching fingerprints.

4.3.2 Generating Estimator Functions

The user does not need to explicitly provide an estimator function. Simple cyclical dependencies between models make it possible to extract an estimator function by fixing one model's output to its value at a given step. Indeed, any Markov function that models an infrequently discontinuous process can be made into a viable estimator by reusing state in a similar way.

A function F_{mkv} defining a Markov process with per-step state P_i generates the next step's state: $F_{mkv}(P_i, Q) = P_{i+1}$. We can define a rudimentary estimator function $F_{est,i}$ by fixing F_{mkv}'s input state at one point in time.

$$F_{est,i}(Q) = F_{mkv}(P_i, Q)$$

Even this rudimentary estimator function can be quite powerful when combined with fingerprints – any uniform changes in state are absorbed by the mapping function.

For example, consider the Markov jump query illustrated in Figure 4.8. The special `CHAIN` parameter type is used to chain the output of one stage of the Markov computation to the following one – in this case chaining the output of `ReleaseWeekModel` to the subsequent `DemandModel` invocation.

As before, `ReleaseWeekModel` has a single discontinuity at the point where `DemandModel`'s output exceeds a certain threshold. Each step in the Markov chain corresponds to predictions for one specific week. The interesting output of this model is `demand`. An estimator from this value will be constructed by fixing `release_week` (the chain parameter) at its initial value. Until the Markov process enters the region of the chain (and after it exits) where the discontinuity is likely to occur, the demand model can be effectively approximated by this

```
-- DEFINITION --
DECLARE PARAMETER @current_week
        AS RANGE 0 TO 52 STEP BY 1;
DECLARE PARAMETER @release_week
        AS CHAIN release_week
        FROM @current_week : @current_week - 1
        INITIAL VALUE 52;
SELECT ReleaseWeekModel(demand) AS release_week, demand
  FROM (SELECT DemandModel(@current_week, @release_week)
                AS demand)
  INTO results
-- BATCH MODE --
...
```

Figure 4.8: A Jigsaw query with a Markovian dependency

non-Markovian estimator.

Recall that normal parameters in Jigsaw are specified in terms of sets or sequences. Each chain parameter is tightly coupled to a non-Markovian parameter, which defines the step identifiers for the process. The `FROM` field of chain parameter definition declares this coupling and states how step identifiers are related. The remaining two fields: `INITIAL VALUE` and `CHAIN` specify an initial value and a query output identifier, respectively. When one step of the query is evaluated, the parameter takes on the corresponding value.

4.4 Interactive What-ifs

Jigsaw's heuristic approach to sampling is ideally suited to the task of online what-if exploration. Moreover, the sort of parameter exploration problems that Jigsaw addresses also benefit from having a human in the loop—imprecise goal conditions that are difficult to specify programmatically can often be reached

easily by a human operator.

A human operator indicates which regions of the parameter space are interesting, and Jigsaw provides progressively more accurate results for that region. Metadata supplementing the simulation query allows Jigsaw to interpret the query results and to produce and progressively refine a graphical representation of the query output for a given set of parameter values.

Unlike its offline counterpart, the goal of online Jigsaw is to rapidly produce accurate metrics for a small set of points in the parameter space. Fingerprinting is used primarily to improve the accuracy of Jigsaw's initial guesses; a very small and quickly generated (e.g., of size 10) fingerprint allows Jigsaw to identify a matching basis distribution and reuse metrics precomputed for it.

Jigsaw operates in an event loop (shown in Algorithm 9), using a *Guide* heuristic to select between the following three categories of processing tasks:

Refinement. Once the initial guess is generated, Jigsaw begins generating further samples for points (i.e., parameter values) of interest. In addition to improving the accuracy of the displayed results, the new samples are used to improve the accuracy of the basis distribution's precomputed metrics.

Validation. Latency also places stringent requirements on the size of fingerprints. Larger fingerprints produce more accurate estimates, but take longer to produce. However, in an online setting, Jigsaw constructs the fingerprint progressively. In addition to generating additional samples for the basis distribution, Jigsaw also reproduces samples for the points of interest that are already present in the basis distribution. The duplicate samples effectively extend the point's fingerprint by validating the existing mapping; if the new points do not

Algorithm 9: $SimplifiedEventLoop(p, State)$

Require: One point of interest p. A lookup table $State[]$ containing, for all points: a mapping function $\mathcal{M}$, the point's fingerprint θ, and the point's basis distribution.

1: **loop**
2: $(\theta, basis, \mathcal{M}) \leftarrow State[p]$; $next \leftarrow p$; $task \leftarrow TaskHeuristic(p)$
3: **if** $task$ = `refinement` **then**
4: $candidate\_ids \leftarrow \{id | id \notin basis\}$
5: **else if** $task$ = `validation` **then**
6: $candidate\_ids \leftarrow \{id | id \in basis \wedge id \notin \theta\}$
7: **else if** $task$ = `exploration` **then**
8: $next \leftarrow ExploreHeuristic(p)$ {Find a nearby point}
9: **if** $State[next].\theta \neq \emptyset$ **then** $candidate\_ids \leftarrow \{id | id \notin State[next].basis\}$
10: **else** $candidate\_ids \leftarrow [0, 10)$ **end if**
11: $sample\_ids \leftarrow PickAtRandom(10, candidate\_ids)$
12: $values \leftarrow EvaluateBlackBox(next, sample\_ids)$
13: $State[next].\theta \leftarrow State[next].\theta \cup values$
14: **if** $State[next].basis \sim_{\mathcal{M}} State[next].\theta$ **then**
15: $(State[next].basis, State[next].\mathcal{M}) \leftarrow FindMatch(State[next].\theta)$
16: **else**
17: $State[next].basis \leftarrow State[next].basis \cup \mathcal{M}^{-1}(values)$

match the values mapped from the basis distribution, Jigsaw finds or creates a new basis distribution.

Exploration. In addition to the above two processing tasks, Jigsaw heuristically

Capacity(current_date, purchase_date_1, purchase_date_2). The Capacity black box simulates a series of purchases. Each purchase increases the capacity of the server cluster after an exponentially distributed delay.

Demand(current_date, feature_release). The Demand black box simulates a simple linearly growing gaussian demand model. As of the feature_release week, the growth rate is changed.

Overload(current_date, purchase_date_1, purchase_date_2). A black box synthesized from Capacity and Demand. Demand's feature_release is ignored, and this black box returns 1 if Demand is greater than Capacity, and 0 otherwise.

UserSelection(current_date). The UserSim black box simulates the per-user requirements of each of a set of users.

SynthBasis(parameter_point). A synthetic black box based on Demand, but with a deterministic number of basis distributions.

MarkovStep(current_date, before_or_after). A simple Markovian process simulating the behavior of Demand with a Markovian dependency introduced between feature_release and the prior date's demand.

MarkovBranch(prior_state). A synthetic black box where at each step, a state counter is incremented by one with a predefined probability. The states diverge at some specified rate.

Figure 4.9: Black boxes used to evaluate Jigsaw

selects points in the parameter space that are likely to be of interest to the user in the near future (e.g., adjacent points in a discrete parameter space). For each point explored, Jigsaw either generates a fingerprint (if none exists), or extends the point's basis distribution with a small number of additional samples.

For clarity, a distinction has been made between samples produced for fingerprints and those produced for basis distributions. However, in most cases there is no difference between either process. For any invertible mapping function, samples are generated directly for the point of interest, and mapped back to the basis distribution by the inverse of the mapping function $\mathcal{M}^{-1}$. For example, for linear mappings $\mathcal{M}(x) = \alpha x + \beta$, the inverse $\mathcal{M}^{-1}(x) = \frac{x-\beta}{\alpha}$.

4.5 Evaluation

Implementations. The original prototype of Jigsaw is implemented as a C# PDB layer built on top of Microsoft SQL Server. The black box functions are implemented as stored procedure written in C#. The C# layer interacts with the SQL Server query execution engine by simply invoking it on subqueries and post-processing the results outside DBMS.

However, this implementation is not well-suited for performance evaluation of Jigsaw, as timing results are polluted by noise from interprocess communication and SQL interpretation and evaluation overheads. In order to achieve a more representative comparison, and to streamline the testing process, a second prototype of Jigsaw query evaluation engine has been constructed entirely in Ruby (without any commercial DBMS). Queries are implemented as black box functions in Ruby, and invoked by a driver process. This simple implementation is representative of how Jigsaw's functionality can be implemented within a probabilistic database's query evaluation engine. These two prototypes are compared in Section 4.5.1.

Black Box Functions. Experiments use a variety of black boxes described in Figure 4.9. Though several synthetic black-boxes are used to identify specific performance characteristics, the Capacity, Demand, Overload, User Selection and Markov Step black boxes are permutations of actual Jigsaw use cases in real cloud infrastructure management scenarios. Specific numbers (i.e., the mean and standard deviation of a normal distribution) have been replaced by ad-hoc values, but the structure of these models remains intact. In all experiments, the entire parameter space for a particular black box is explored.

Experimental Setup. Experiments are performed on a 2.4 GHz Core2 Duo with 4 GB of RAM. Except where stated, experiments assume a need for exactly 1000 sample instances per point in the parameter space, and use a fingerprint size of 10. Results shown are the average of 30 trials.

4.5.1 Comparison of Two Prototypes

Figure 4.10 shows a brief comparison of the relative performance of Jigsaw's C# + MS SQL implementation and the lightweight Ruby engine. As shown, for simple data-independent queries, the Ruby implementation is able to achieve much better performance, as the dominant cost is that of invoking the black box, rather than the overheads of repeatedly invoking the query processor. The one case where the Ruby implementation is not representative of the offline engine is black boxes that are heavily data dependent, as in the UserSelection simulation. As might be expected, a database engine's ability to manage large datasets is superior to that of Ruby.

The rest of the experiments in this section use less-data dependent black boxes, and hence the Ruby implementation is used. However, relative performance gains demonstrated by the Ruby prototype are roughly similar to those in the C# + MS SQL implementation.

4.5.2 Baseline Performance

The standalone performance gain of fingerprinting are compared against a naive-generate everything approach in Figure 4.5.1. The figure shows timing results

Model	Online Speed	Offline Speed
Demand	0.1964 s/pc	0.00096 s/pc
Capacity	0.84525 s/pc	0.0028 s/pc
Overload	5.4625 s/pc	0.092825 s/pc
UserSelect	34.4 s/pc	252.454 s/pc

Figure 4.10: User Interface Wrapper vs Core Engine Simulator Timing comparison. Values are in time per parameter combination.

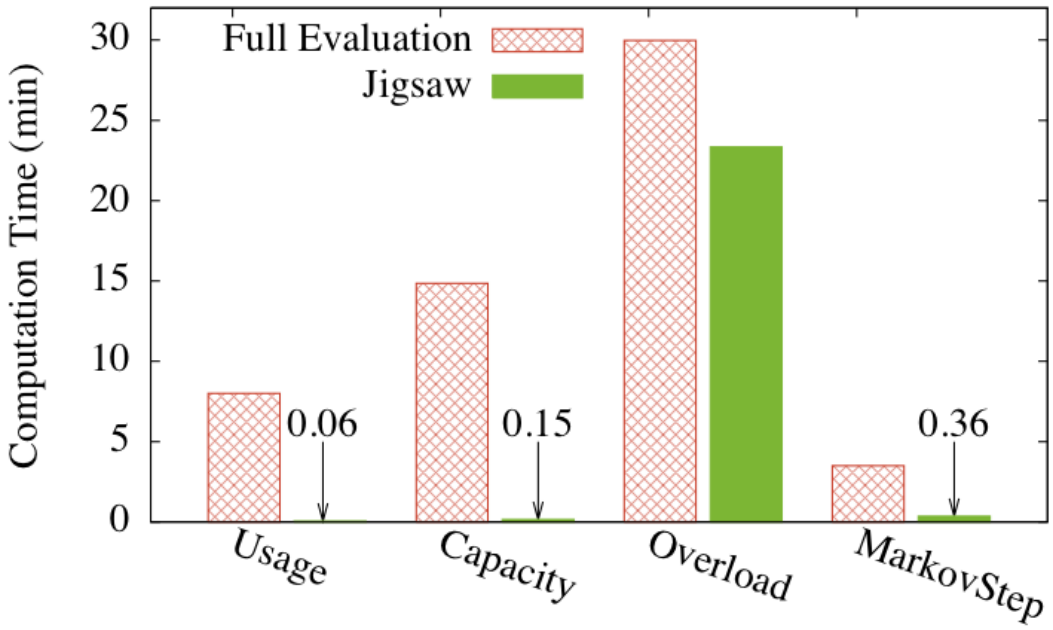


Figure 4.11: Jigsaw vs fully exploring the parameter space.

for several queries, each evaluated both with and without fingerprinting. The extremely simplistic *Demand* model requires only one basis distribution for its entire ~5000 point parameter space, and can be evaluated almost instantaneously. Even relatively complex event-based models like *Capacity* (which has a parameter space of ~8000 points) and *MarkovStep* (evaluated over ~2500 steps) require only a few basis distributions.

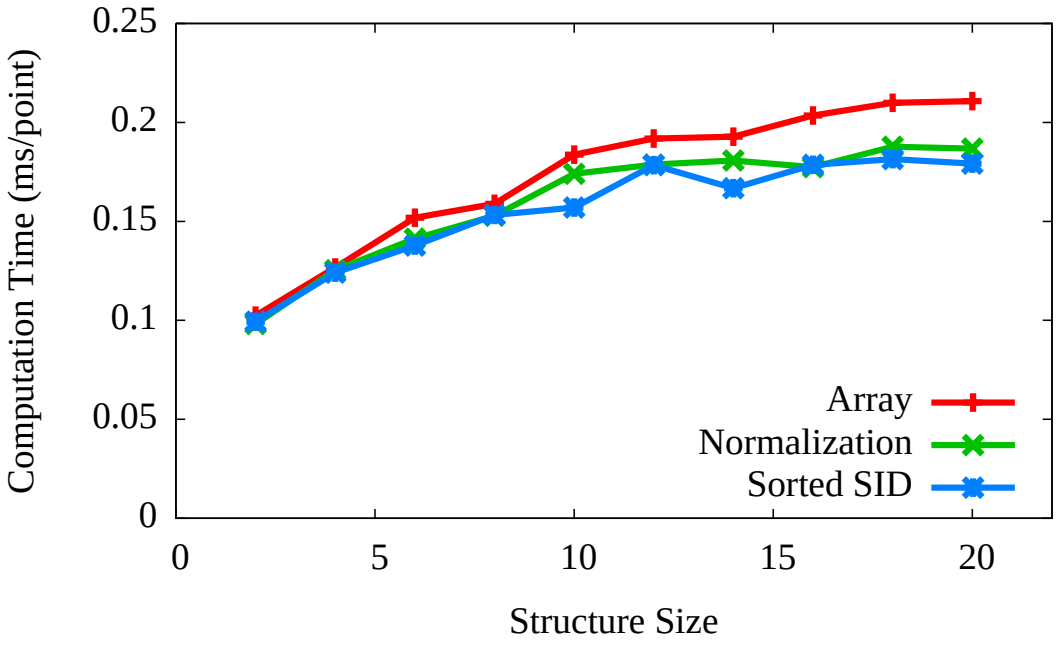


Figure 4.12: Computation time versus the size of structures in the Capacity model

Overload is an interesting case: despite being defined as a query over two two black boxes for which Jigsaw can provide a substantial performance boost (compare Overload with ~8000 points and Demand and Capacity's timing), the joint query is only computed in half the time. The reason for this is that the query produces a boolean result: the output of a comparison between two values, and information about the two original values is lost. Effectively, Jigsaw is unable to reuse basis values by re-mapping them. This strongly suggests that Jigsaw's techniques can be further improved by incorporating them into a database engine with a symbolic execution strategy such as PIP. In such a system, database operations between random variables (i.e., VG-Function-generated values) mapped from the same basis distribution are resolved symbolically. For example, consider two random variables X, Y such that $X = \mathcal{M}_X(f(x)) =$

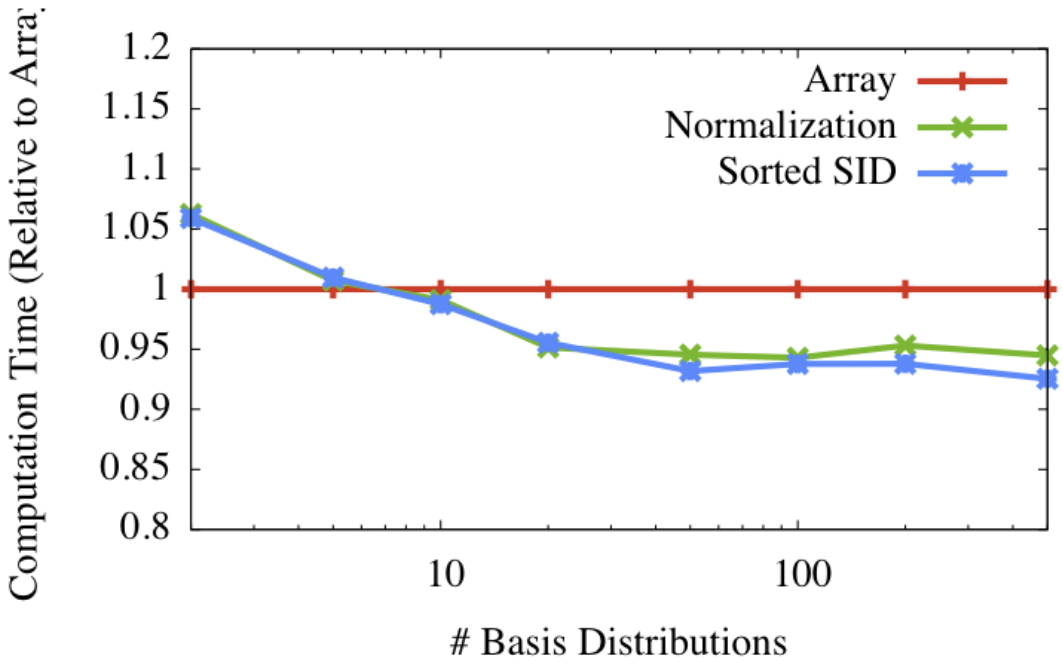


Figure 4.13: Indexing in a static parameter space. Note the range on the y-axis – from 0.8 to 1.2 times the relative performance of naive array-scan.

$2 \cdot f(x) + 2$ and $\mathcal{M}_Y(f(x)) = 3 \cdot f(x) + 3$. Jigsaw can symbolically produce $X + Y = (\mathcal{M}_X + \mathcal{M}_Y)(f(x)) = 5 \cdot f(x) + 5$. Similarly, given a histogram of $f(x)$, Jigsaw can efficiently compute the probability that $M_X(f(x)) > M_Y(f(x))$.

The *Capacity* model also deserves more discussion. As discussed in Section 4.1, the model produces a line with several non-localized discontinuities or *structures* – one for each purchase. However, each of the discontinuities is surrounded by a structure spanning a range of dates: each purchase is followed by a short period during which the simulated hardware has not come online in a(n exponentially shrinking) fraction of the sample instances. Figure 4.5.1 relates the number of basis distributions to the size of each structure (the number of "weeks" that it spans). Note that the relationship between structure size grows

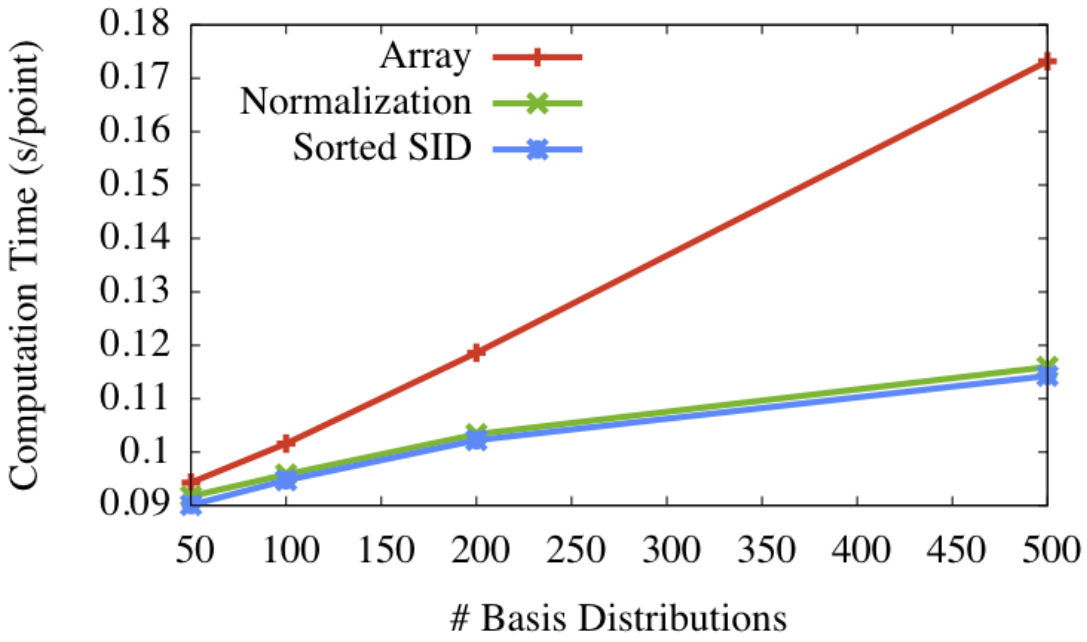


Figure 4.14: Indexing, growing the parameter space with basis size.

and the number of basis distributions is sub-linear. Jigsaw is able to recognize individual points in each structure (i.e., four weeks after one purchase, and the week of the second purchase), and reuse the corresponding basis distribution.

Accuracy. In theory, the principle of using fingerprints can introduce two sources of errors in a general simulation framework. The first source is the possibility of selecting an incorrect fingerprint due to insufficient fingerprint length. Significant error of this sort was not observed in any of the experiments, suggesting that a fingerprint length of 10 is sufficient for the models considered.

In online mode, Jigsaw continually validates mappings between points in its parameter space by randomly generating samples from a mapped distribution that have already been obtained from the mapping – the size of the fingerprint

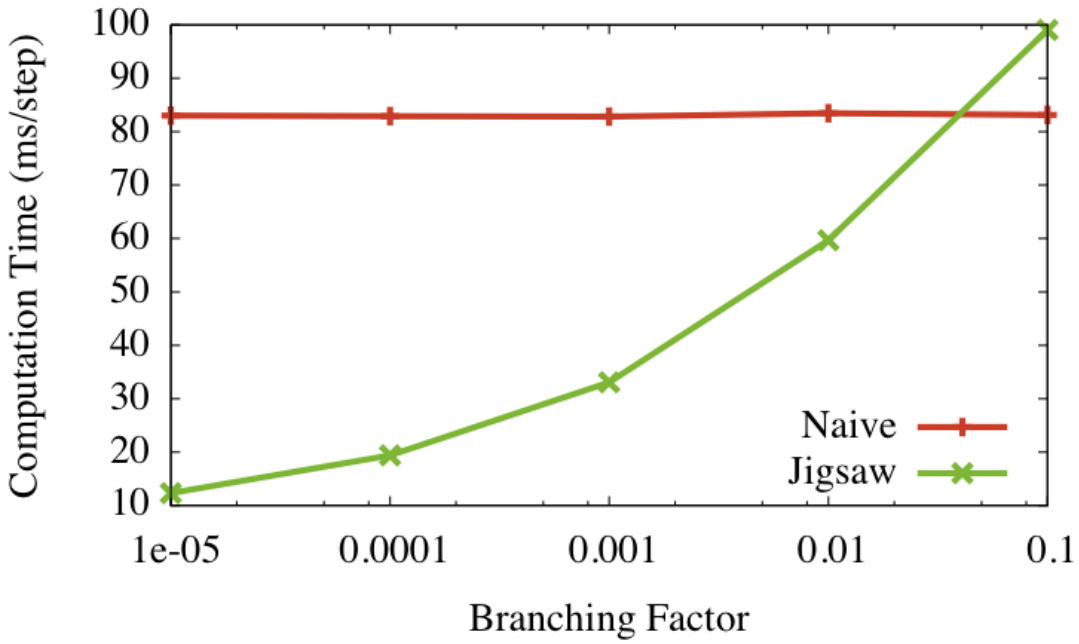


Figure 4.15: Performance for a Markov process.

grows over time in online mode. A similar, random check can be instantiated in Jigsaw's offline mode, dynamically growing the fingerprint size for all distributions if any mapping errors are detected.

The second source of possible error is due to correlation of results under different parameter values (since the same random seed is used across parameter values). However, Jigsaw never combines such correlated results of different parameters—it only compares them. Hence we do not see such errors in our results. In other words, outputs of Jigsaw are equivalent to full simulation for each possible parameter value.

4.5.3 Indexing

Next, the behavior of the two indexing strategies described in Section 4.2: *Normalization*, and *Sorted SID* indexing are considered. In this test, black boxes were synthesized to generate a specific number of basis distributions. The expectation of each black-box was computed for 1000 different parameter combinations. Figure 4.13 shows the performance of each indexing strategy relative to performing a naive *Array* scan for each lookup. The overall results are not surprising: The costs of an array scan begin to dominate the static costs of indexing as the basis grows past 50 elements.

Both indexing schemes perform substantially better than naive array scan, with Normalization trailing behind Sorted SID slightly. After a basis size of about 200 (where full sample generation is required for 20% of the parameter space) is reached, the cost of sample generation begins to dominate the cost of basis matching. Indexing continues to asymptotically approach a 10% reduction in computation time. This is best illustrated in Figure 4.14. Here, the size of the parameter space (and consequently the total amount of computation) is scaled relative to the basis size. The basis size is fixed at 10% of the parameter space. As expected, naive Array scan scales linearly with basis size, while the indexing strategies scale sub-linearly.

4.5.4 Markovian Jumps

Next, Jigsaw's performance on Markov processes is analyzed. Markov processes consisting of periodic, but infrequent discontinuities are ideal for Jigsaw; this sort of black box behavior generates frequently overlapping states and ad-

mits an easy estimator, which simply assumes that the state stays the same. Such a process has already been illustrated in Figure 4.5.1.

The benefits and limitations of Jigsaw on more complex, diverging models are also of interest. Figure 4.15 shows Jigsaw's performance on a black-box Markovian process synthetically generated to diverge at a predefined rate. In this figure, the term branching refers to the probability of divergence at each timestep. The black box was invoked for 128 steps, and Jigsaw attempted to accelerate evaluation by skipping ahead in the process.

This shows Jigsaw's applicability to Markovian processes characterized by periodic discontinuities. Even in its default configuration, Jigsaw is able to improve the efficiency of Markovian processes where as many as one in twenty steps involves a discontinuity. Indexing strategies designed specifically for Markovian processes (e.g., discard all basis values except the last), can improve this even further.

CHAPTER 5

DBTOASTER

This chapter describes DBToaster, a Dynamic Database Management System that supports complex high-rate data monitoring tasks by taking a novel approach to view maintenance referred to as agile views. Generated as the result of a recursive query compilation process, agile views can aggressively maintain query state as incrementally as possible – ensuring that they are able to process complex, stateful queries over extremely rapidly changing data. This sort of automated construction of efficient monitoring systems finds applications across a broad swath of fields from network monitoring, to scientific experimentation, to the stock market.

DBToaster is being developed in collaboration with Yanif Ahmad (now at Johns Hopkins) and my advisor Christoph Koch. Former and more recent collaborators on the DBToaster project include Anton Mozorov and Aleksandar Vitorovich. Portions of the content presented were originally presented at CIDR 2011 [49]

The DBToaster project was supported by the US National Science Foundation under grant IIS-0911036. Any opinions, findings, conclusions or recommendations expressed are those of the authors and do not necessarily reflect the views of NSF.

5.1 The Architecture of a DDMS

DBToaster is an instantiation of a Dynamic Data Management System (DDMS). DDMS are a new class of database systems built to support complex, albeit rarely changing query workloads constructed over frequently changing datasets. Concretely, the design of a DDMS is based around four criteria:

1. The stored dataset is large and changes frequently.
2. The maintenance of materialized views dominates ad-hoc querying.

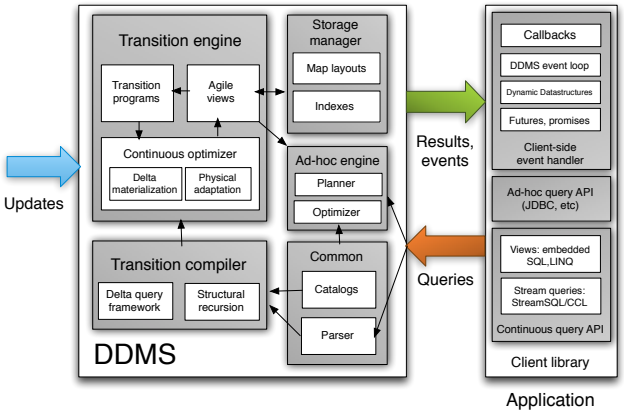


Figure 5.1: Dynamic Data Management System (DDMS) and Application Interface Architecture

3. Access to the data is primarily by monitoring the views and performing simple computations on top of them.

   Some updates cause events, observable in the views, that trigger subsequent computations, but it is rare that the data store is accessed asynchronously by humans or applications.

4. Updates happen primarily through an *update stream*. Computations triggered by view events usually do not cause updates: there is usually no feedback loop.

A DDMS is a lightweight system that provides large dynamic data structures to support declarative views of data. A DDMS is *agile*, keeping internally maintained views fresh in the face of dynamic data. Client applications primarily interact with a DDMS by registering callbacks for view changes, rather than by accessing views directly. A DDMS does not necessarily provide additional DBMS functionality such as persistency, transactions, or recoverability.

The architecture of a DDMS is illustrated in Figure 5.1. The core component of a DDMS is its runtime engine. Unlike a traditional database system where the same engine manages all database instances, each individual DDMS execution runtime is constructed around a specific set of queries provided by the client program (e.g., via SQL code embedded inline in the program), each defining an *agile view*.

5.1.1 Application Interfaces

The data that is processed by a DDMS arrives at the system in the form of an update stream of tuple insertions, deletions and modifications. The stream need not be ordered in any shape or form, and deletions are assumed to apply to tuples that have already been seen at some arbitrary prior point on the stream. Updates are fully processed on-the-fly, and their effects on agile views are realized in atomic fashion, prior to working on any subsequent update. Depending on the type of results requested by queries, any results arising from updates will be directly forwarded to application code as agile views are maintained.

DBToaster provides a wide variety of client interfaces to issue queries and obtain results from the DDMS, to reflect the diverse needs of applications built on top of it. Today's stream processors tend to be black-box systems that run completely decoupled from the application. Client libraries interact with stream processors through remote procedure call abstractions, issuing queries and new data through function calls, and either polling or being notified whenever results appear on a queue that is associated with a TCP socket connected to the stream processor.

In DBToaster, the set of agile views requested by clients, the *visible schema*, forms the primary read interface between client programs and the DDMS runtime. Clients can submit queries for which the DDMS materializes an agile view through three methods:

1. An embedded language, whose syntax and data model are natural fits to the host language in which the client application is written. Examples include embedded SQL, and collection comprehension oriented approaches such as LINQ, Links, and Ferry [24, 35, 65]. One interesting challenge with the embedded language approach is that of enabling asynchronous event-driven programming. Whereas language embeddings are natural for ad-hoc querying, we have yet to see these approaches for stream processing. This is the main mode of specifying queries.
2. A continuous query client API, as done with existing stream client libraries, which sends a query string to the DDMS server for parsing, compilation, and agile view construction. The query string may be specified in a standard streaming language similar to StreamSQL or CCL [45]. The client may specify several ways to receive results, as seen below.
3. An ad-hoc query client API, which issues a one-time query to the DDMS, and returns the agile view as a data-structure to be used by the remainder of the client program. This API may be used in both synchronous and asynchronous modes, as indicated by the type of result requested. The query is specified in standard SQL.

Given these modes of issuing queries to the DDMS, the DBToaster client interface supports four methods of receiving results:

1. Callbacks, which can be specified as handlers as part of the continuous query API. Callbacks receive a stream of query results, and are the simplest form of result handlers that run to completion on query result events.
2. A DDMS event loop, which multiplexes result streams for multiple queries. Applications may register callbacks to be executed on any result observed on the event loop, allowing complex application behavior through dynamic registration, observation and processing of results on the event loop.
3. Dynamic data-structures, which are read-only from the application perspective. The data-structure appears as a native collection type in the host language, facilitating natural access for the remainder of the program. Ad-hoc queries use this method for results by default. Continuous queries may also use this method in which case the data-structure acts as a proxy with accessors that pull in any updates from the DDMS when invoked.
4. Promises and futures [63], which provide a push-based proxy data-structure for the result. A future is an object whose value is not initially known and is provided at a later time. A program using a query returning a future can use the future as a native datatype, in essence constructing a client-side dataflow to be executed whenever the future's value is bound. In our case, this occurs whenever query results arrive from the DDMS. Language embedded stream processing can be supported by futures, or program transformations to construct client side dataflow, such as continuation passing style as found in the programming languages literature [83].

5.1.2 DDMS Internals

The internals of the runtime engine itself are best viewed through the lens of a state machine. Compared to similar abstractions for complex event processors [5, 16], the state is substantially larger. Conceptually, the state represents an entire relational database and transitions represent changes in the base relations: events in the update stream.

Compiling transitions. Each transition causes maintenance work for agile views, and just as with incremental view maintenance this work can be expressed as queries. Maintenance can be aided by dynamic data structures, that is, additional agile views making up an *auxiliary schema*. A DDMS is a long-running system, operating on a finite number of update streams.

This combination of characteristics naturally suggests *compiling* and specializing the runtime for each transition and associated maintenance performed by a DDMS. The transition compiler generates lightweight transition programs that can be invoked by the runtime engine with minimal overhead on the arrival of events. The compiler is discussed in further detail in Section 5.2.

Storage management and ad-hoc query processing. Given the instantiation of an auxiliary schema and agile views, a DDMS must intelligently manage memory utilization, and the memory-disk boundary as needed. The storage manager of a DDMS is responsible for the efficient representation of both the agile views and any index structures required on these views. Section 5.4 discusses the issue of indexing, as well as how views are laid out onto disk. Supporting ad-hoc query processing turns out to be relatively straightforward given that the core of a DDMS continuously maintains agile views. Ad-hoc queries can be

rewritten to use agile views in a similar fashion to the materialized view usage problem in standard query optimization. A key challenge here is how to ensure consistency, such that ad-hoc queries do not use inconsistent agile views as updates stream in and the DDMS performs maintenance. On the other hand, ad-hoc queries should not block the DDMS' maintenance process and incur result delivery latency for continuous queries.

One option here is to maintain a list of undo actions for each ad-hoc query with respect to agile view maintenance. This design is motivated by the fact that continuous queries are the dominant mode of usage, and ad-hoc queries are expected to occur infrequently, thus the concurrency control burden falls on the executor of ad-hoc queries.

Runtime adaptivity. Significant improvements in just-in-time (JIT) compilation techniques mean that transition programs need not be rigid throughout the system's lifetime. A DDMS includes a compiler and optimizer working in harmony, leveraging update stream statistics to guide the decisions to be made across the database schema, state and storage.

For example, the compiler may choose to compute one or more views on the fly, rather than maintaining them in order to keep expected space usage within predefined bounds. The optimizer's decisions are made in terms of the space being used, the cost of applying transitions on updates, as well as information from a storage manager that aids in physical aspects of handling large states, including implementing a variety of layouts and indexes to facilitate processing.

5.2 Realizing Agile Views

Agile views are database views that are maintained as incrementally as possible. Despite more than three decades of research into incremental view maintenance (IVM) techniques [34, 76, 93, 94], agile views have not been realized – one of the key challenges in handling large dynamic datasets is identifying exploiting further opportunities for incremental computation during maintenance.

Conceptually, current IVM techniques use delta queries for maintenance. However, observe that the delta query is itself a relational query that is amenable to incremental computation. The delta queries can be materialized as auxiliary views, and so forth, recursively maintaining both the visible and auxiliary views.

Furthermore, repeated delta transformations successively simplify queries.

5.2.1 View Maintenance in DBToaster

Given a query q defining a view, IVM yields a pair $\langle m, Q' \rangle$, where m is the materialization of q, and Q' is a set of delta queries responsible for maintaining m (one for each relation used in q that may be updated). DBToaster makes the following insight regarding IVM: current IVM algorithms evaluate a delta query entirely from scratch on every update to any relation in q, using standard query processing techniques. DBToaster exploits that a delta query q' from set Q' can be incrementally computed using the same principles as for the view query q, rather than evaluated in full.

To convey the essence of the concept, IVM takes q, produces $\langle m, Q' \rangle$ and performs m += $q'(u)$ at runtime, where u is an update to a relation R and q' is the delta query for updates to R in Q'. This is one step of *delta (query) compilation*. This is the extent of query transformations applied by IVM for incremental processing of updates.

DBToaster applies this concept *recursively*, transforming queries to *higher-level deltas*. DBToaster starts with q, produces $\langle m, Q' \rangle$ and then recurs, taking each q' to produce $\langle m', Q'' \rangle$ and repeating. Here, each m' is maintained as m' += $q''(v)$, where v is also an update, (possibly) different from u above, and q'' is the delta query from Q'' for the relation being updated. q' and q'' are referred to as first- and second-level delta queries respectively. DBToaster again recurs for each q'', materializing it as m'', and maintaining it using third-level queries Q''', and so forth.

While delta queries are relational queries, they have certain characteristics that facilitate recursive delta compilation. First, DBToaster delta queries are parameterized SQL queries. That is, the query is defined with certain variables in the query bound outside of the query itself. If a parameter can also be bound inside the query, it effectively becomes a group-by term – DBToaster materializes and stores query results for the entire domain of the parameter. If the parameter is entirely unbound within the query, then the view acts as a sort of self-maintaining cache – newly encountered parameter values trigger a partial runtime evaluation of the query, and DBToaster incrementally maintains these results as if they were group by terms. This concept is discussed further below.

Thus, in particular, higher-level deltas are just (parameterized) SQL queries, but are not *higher-order* in the sense of functional programming, as some queries

in complex-value query languages are [17].

Example 5.2.1 *To illustrate parameters, consider one step of delta compilation on the following query q over a schema* `R(a int, b int), S(b int, c int)`*:*

$$q = \texttt{SELECT sum(a*c) FROM R NATURAL JOIN S}$$

For an update u that is an insertion of tuple ⟨`@a`, `@b`⟩ *into relation R, the delta for q is:*

$$\begin{aligned} q_R = \Delta_u(q) = {} & \texttt{SELECT sum(@a*c) FROM values(@a,@b), S} \\ & \texttt{WHERE S.b = @b} \\ = {} & \texttt{@a*(SELECT sum(c) FROM S WHERE S.b=@b)} \end{aligned}$$

The `values (...)` *clause is PostgreSQL syntax for a singleton relation defined in the query. Transforming a query into its delta form for an update u on R introduces parameters in place of R's attributes. We also apply a rewrite exploiting distributivity of addition and multiplication to factor out parameter* `@a` *from the query.*

The second property, key to making recursive delta processing feasible is that, for a large class of queries, delta queries are structurally strictly simpler than the queries that the delta queries are taken off.

This can be made precise as follows. Consider SQL queries that are sum-aggregates over positive relational algebra. Consider positive relational algebra queries as unions of select-project-join (SPJ) queries. The *degree* of an SPJ query is the number of relations joined together in it. The degree of a positive relational algebra query is the maximum of the degrees of its member SPJ queries and the degree of an aggregate query is the degree of its positive relational algebra component.

The rationale for such a formalization – based on viewing queries as polynomials over relation variables – is discussed in detail in [55]. It is proven in that paper that the delta query of a query of degree k is of degree $\max(k-1, 0)$. A delta query of degree 0 only depends on the update but not on the database relations. So DBToaster guarantees that a k-th level delta query $q^{(k)}$ has lower degree than a *(k-1)*-th level query $q^{(k-1)}$. Recursive compilation terminates when all conjuncts have degree zero.

Example 5.2.2 *Consider the delta query q_R above, which is of degree 1 while q is of degree 2. Query q_R is simpler than q since it does not contain the relation R. This point is further illustrated by looking at a recursive compilation step on q_R. The second compilation step materializes q_R as:*

$$m_R = \texttt{SELECT sum(c) FROM S WHERE S.b=@b}$$

omitting the parameter `@a` *since it is independent of the above view definition query. DBToaster can incrementally maintain m_R with the following delta query on an update v that is an insertion of tuple $\langle$`@c`, `@d`$\rangle$ into relation S:*

$$q_{RS} = \Delta_v(q_R) = \texttt{SELECT @c FROM values(@c,@d)}$$

The delta query q_{RS} above has degree zero since its conjuncts contain no relations, indeed the query only consists of parameters. Thus recursive delta compilation terminates after two rounds on query q.

5.2.2 Agile Views

DBToaster materializes higher-level deltas as agile views for high-frequency update applications with continuous group-by aggregate query workloads. Agile

Input (parent query)		Update	Output: auxiliary map, delta query		
q =	`SELECT l.ordkey, o.sprior, sum(l.extprice) FROM Customer c, Orders o, Lineitem l WHERE c.custkey = o.custkey AND l.ordkey = o.ordkey GROUP BY l.ordkey, o.sprior;`	`+Customer (ck,nm,nk,bal)`	m[][*ordkey*, *sprior*]	q_c =	`SELECT l.ordkey, o.sprior, sum(l.extprice) FROM Orders o, Lineitem l WHERE @ck = o.custkey AND l.ordkey = o.ordkey GROUP BY l.ordkey, o.sprior;`
q_c:	Recursive call, see previous output	`+Lineitem (ok,ep)`	m_c[][*custkey*, *ordkey*, *sprior*]	q_{cl} =	`SELECT @ok, o.sprior,@ep*sum(1) FROM Orders o WHERE @ck = o.custkey AND @ok = o.ordkey`
q_{cl}:	Recursive call, see previous output	`+Order (ck2,ok2,sp)`	m_{cl}[][*custkey*, *ordkey*, *sprior*]	q_{clo} =	`SELECT @sp, count() WHERE @ck = @ck2 AND @ok = @ok2;`

Figure 5.2: Recursive query compilation in DBToaster. For query q, we produce a sequence of materializations and delta queries for maintenance: $\langle m, q' \rangle, \langle m', q'' \rangle, \langle m'', q''' \rangle$. This is a partial compilation trace, our algorithm considers all permutations of updates.

views are represented as main memory (associative) map data structures with two sets of keys (that is a doubly-indexed map $m[\vec{x}][\vec{y}]$), where the keys can be explained in terms of the delta query defining the map.

Recall that delta queries are parameterized SQL queries. The first set of keys (the *input* keys) correspond to the parameters, and the second set (the *output* keys) to the select-list of the defining query. In the event that a parameter appears in an equality predicate with a regular attribute, we omit it from the input keys because we can unify the parameter. Other interesting manipulations of parameterized queries are discussed in Section 5.4.

Example 5.2.3 *Figure 5.2 shows the compilation of a query q:*

```
   SELECT l.ordkey, o.sprior, sum(l.extprice)
     FROM Customer c, Orders o, Lineitem l
    WHERE c.custkey = o.custkey and l.ordkey = o.ordkey
 GROUP BY l.ordkey, o.sprior
```

inspired by TPC-H Query 3, with a simplified schema:

Customer(custkey,name,nationkey,acctbal)

Lineitem(ordkey,extprice)

Order(custkey,ordkey,sprior)

The first step of delta compilation on q produces a map m. The aggregate for each group ⟨*ordkey*, *sprior*⟩ *can be accessed as m*[][*ordkey*, *sprior*]. *DBToaster can answer query q by iterating over all entries (groups) in map m, and yielding the associated aggregate value. The first step also computes a delta query* q_c *by applying standard delta transformations as defined in existing IVM literature [34, 76, 93, 94]. In summary, these approaches substitute a base relation in a query with the contents of an update, and rewrite the query. For example, on an insertion to the* `Customer` *relation, this relation is substituted with an update tuple* ⟨`@ck,@nm,@nk,@bal`⟩*:*

```
  SELECT l.ordkey, o.sprior, sum(l.extprice)
    FROM values (@ck,@nm,@nk,@bal)
           AS c(custkey,name,nationkey,acctbal),
         Orders o, Lineitem l
   WHERE c.custkey = o.custkey and l.ordkey = o.ordkey
GROUP BY l.ordkey, o.sprior
```

Above the substitution replaces the `Customer` *relation with a singleton set consisting of an update tuple with its fields as parameters. The resultant query* q_c *can be simplified as:*

q_c =
```
SELECT l.ordkey, o.sprior,sum(l.extprice)
  FROM Orders o, Lineitem l
 WHERE @ck = o.custkey
   AND l.ordkey = o.ordkey
GROUP BY l.ordkey, o.sprior;
```

The query rewrite replaces instances of attributes with parameters through variable substitution, as well as more generally (albeit not seen in this example for simpler exposition of the core concept of recursive delta compilation), exploiting unification, and distributivity properties of joins and sum aggregates to factorize queries [55].

This completes one step of delta compilation. The compilation algorithm also computes deltas to q for insertions to `Order` *or* `Lineitem` *(i.e.* q_o *and* q_l*). The full transition program for all insertions is shown in Figure 5.3, while deletions are symmetric.*

IVM techniques evaluate q_c *on every insertion to* `Customer`*. To illustrate the recursive nature of the DBToaster compilation process, consider one possible subsequent step: compilation of* q_c *to* m_c, q_{cl} *on an insertion to* `Lineitem` *(see the second row of Figure 5.2). At this second step, DBToaster materializes* q_c *with its parameter* `@ck` *and group-by fields as* $m_c[][custkey, ordkey, sprior]$*, and uses this map* m_c *to maintain the query view m:*

```
on_insert_customer(ck,nm,nk,bal):
  m[][ordkey,sprior] += m_c[][ck,ordkey,sprior];
```

As it turns out, all maps instantiated from simple equijoin aggregate queries such as TPCH Query 3 have no input keys. Maps with input keys only occur as a result of inequality predicates and correlated subqueries, for example the VWAP query from Ex-

```
on_insert_customer(ck,nm,nk,bal) :
  m[][ordkey, sprior] +=
                   m_c[][ck, ordkey, sprior];
  m_l[][ordkey, sprior] +=
                  m_cl[][ck, ordkey, sprior];
  m_o[][ck] += 1;

on_insert_lineitem(ok,ep) :
  m[][ok, sprior] += ep *  m_l[][ok, sprior];
  m_c[][custkey, ok, sprior] +=
           ep *  m_cl[][custkey, ok, sprior];
  m_co[][ok] += ep;

on_insert_order(ck,ok,sp) :
  m[][ok, sp]         += m_co[][ok] * m_o[][ck];
  m_l[][ok, sp]       +=  m_o[][ck];
  m_c[][ck, ok, sp]   += m_co[][ok];
  m_cl[][ck, ok, sp] += 1;
```

Figure 5.3: Trigger functions generated by DBToaster for the query q, for all possible insertion orderings. The path taken through the compilation algorithm is expressed as part of the map name as seen for m_c and m_{cl} in the example walkthrough.

ample 1.2.1.

The above trigger statement in a C-style language fires on insertions to the `Customer` *relation, and describes the maintenance of m by reading the entry* m_c[`ck`, *ordkey*, *sprior*] *instead of evaluating* q_c(`ck`, *Orders*, *Lineitem*). *Notice that the trigger arguments do not contain ordkey or sprior, so where are these variables defined? In DBToaster, this statement implicitly performs an iteration over the domain of the map being updated. That is, map m is updated by looping over all* ⟨*ordkey*, *sprior*⟩ *entries in its domain, invoking lookups on* m_c *for each entry and the trigger argument* `ck`. *Map read and write locations are often (and for a large class of queries, always) in one-to-one correspondence, allowing for an embarrassingly parallel implementation (see Section 5.5).*

For clarity, the verbose form of the statement is:

```
on_insert_customer(ck,nm,nk,bal):
  for each ordkey,sprior in m:
    m[][ordkey,sprior] += m_c[][ck,ordkey,sprior];
```

Henceforth, the implicit loop form will be used. Furthermore, this statement is never implemented as a loop, but rather as a single-instruction, multiple-data (SIMD) operation. Consequently, map data structures must support slicing, or partial-key lookups – in this example, m_c must be able to efficiently produce all tuples $\langle ordkey, sprior \rangle$ matching a given `ck`*. This is trivially implemented with secondary indexes for each partial access present in any maintenance statement.*

This form of maintenance statement is similar in structure to the concept of marginalization in probability distributions, essentially the map m is a marginalization of map m_c over the attribute `ck`*, for each* `ck` *seen on the update stream.*

Returning to the delta q_{cl} produced by the second step of compilation, its derivation and simplification is presented below.

```
SELECT l.ordkey, o.sprior,
       sum(l.extprice)
  FROM Orders o, values
       (@ok,@ep) as
       l(ordkey,extprice)
 WHERE @ck = o.custkey
   AND l.ordkey = o.ordkey
```

⇒

```
SELECT @ok, o.sprior,
       @ep*sum(1)
  FROM Orders o
 WHERE @ck = o.custkey
   AND @ok = o.ordkey
```

Notice that q_{cl} has a parameter `@ck` *in addition to the substituted relation* `Lineitem`*. This parameter originates from the attribute* `c.custkey` *in q, highlighting that map*

parameters can be passed through multiple levels of compilation.

The delta q_{cl} is used to maintain the map m_c on insertions to `Lineitem`*, and is materialized in the third step of compilation as m_{cl}*[][*custkey*, *ordkey*, *sprior*]*. The resulting maintenance code for m_c is (corresponding to Line 8 of the full listing):*

```
on_insert_lineitem(ok,ep) :
  m_c[][custkey, ok, sprior] +=
    ep * m_cl[][custkey, ok, sprior];
```

The above statement indicates an iteration over each ⟨custkey, sprior⟩ pair in the map m_c, for the given value of the trigger argument `ok`*. Again, recall that this corresponds to to a slice access of m_{cl}, of ⟨custkey, sprior⟩ pairs for a given* `ok`*. The third step of recursion on insertion to* `Order` *is the terminal step, as can be seen on inspection of the delta query q_{clo}:*

```
SELECT o.sprior, count()
  FROM values (@ck2,@ok2,@sp)
    AS o(custkey,ordkey,sprior)
 WHERE @ck = o.custkey
   AND @ok = o.ordkey
```

⇒

```
SELECT @sp, count()
 WHERE @ck = @ck2
   AND @ok = @ok2
```

In the result of the simplification, the delta q_{clo} does not depend on the database since it contains no relations, only parameters. Thus the map m_{cl} can be maintained entirely in terms of trigger arguments and map keys alone. Note this delta contains parameter equalities. These predicates constrain iterations over map domains, for example the maintenance code for q_{clo} would be rewritten as:

```
on_insert_order(ck2,ok2,sp) :
  m_cl[][ck, ok, sp] +=
    if ck==ck2 && ok==ok2
    then 1 else 0;
```

⇒

```
m_cl[][ck2, ok2, sp]
  += 1;
```

where, rather than looping over map m_{cl}'s domain and testing the predicates, only the map entry corresponding to `ck2, ok2` *from the trigger arguments are updated.*

This process is subsequently repeated for all other possible insertion orderings, as shown in Figure 5.3. Note that some paths through the compilation produce maps based on equivalent queries; DBToaster detects these and reuses the same map. Also note that for this query, the trigger functions for deletions are symmetric with their insertion counterparts. This sort of symmetry appears in all queries without parameterized subqueries.

5.3 The DBToaster Compiler

DBToaster is built around a relational calculus based on a simplified version of the one presented in [55]. The key insight of this calculus is that instead of representing a query as a set of predicates, set of computations, and set of input sources, it can be represented as a monad [61] – specifically as a composition of operators which compute the arity of elements in the output set. Relational operators are thus transformed into ordinary arithmetic operators over arities (with some caveats described further in [55]): `UNION` becomes a sum, while `NATURAL JOIN` becomes a product.

Note also that this representation is an algebraic ring, which guarantees that the *delta* operation (described below) is closed, and produces a simpler expres-

sion.

The DBToaster relational calculus is summarized in Figure 5.4. Every expression in this calculus is defined by a schema, which includes a set of named input variables – defining the parameters to the expression, and a set of output variables – defining the values of individual columns in result tuples. As before, we write $m[iv][ov]$ to define an expression with a schema consisting of input variables iv and output variables ov.

Note that although the calculus of [55] expresses both input and output variables as parameters to the monad defined by the expression – something which factors deeply into the design of the calculus, it is relevant to note that the expression will be nonzero for only fixed set of valuations of the output variables – the monad defines a set. It is thus useful to define the set of tuples produced by the expression as those valuations of the output variables for which the monad defined by the expression produces a nonzero arity. The expression itself can be thought of as an implicit loop over all of the tuples (and their corresponding arities) that it defines. In summary, an expression in DBToaster's relational calculus is (in effect) a query parameterized by a set of input variables, which produces a set of tuples with a schema corresponding to the output variables.

The leaf terms of DBToaster's relational calculus consist of `Value`s, `Relation`s and `External`s.

`Value` represent the null relation with a single null tuple of the specified arity; If the value is a constant, the constant defines the arity. If the value is a variable, then the variable becomes part of the expression's schema (as an input variable) and the null tuple's arity is equal to the variable's value.

A `Relation` represents the set of tuples contained within one of the base relations being queried – the leaf term is defined with the relation's name, and a set of variables, which define the schema (as output variables) of the term. Note that the contents of the relation are not returned directly – the monad defined by the term produces a nonzero output for those output variable valuations that correspond to tuples in the relation. Another way of looking at it is that, while the `Relation` defines a set, in the arithmetic of a DBToaster calculus expression the term's value is the arity of a tuple in the set. The contents of the tuple are made available contextually via the term's output variables.

Like a `Relation`, an `External` represents a set of tuples, save that it is an internal component of the DBToaster compiler. Over the course of compilation, components of the query will be materialized into subviews, each defined by a subquery. An `External` represents not only the set of tuples described by this subquery, but also includes metadata describing the way the set should be accessed, managed, and maintained.

The two mechanisms for combining subexpressions together are `Sum` and `Prod`uct.

`Sum` corresponds to a union between two or more subexpressions. Although not strictly necessary, for simplicity, it is considered to be an error if these subexpressions have different output variables in their schemas. The subexpressions may contain different input variables – in which case the schema of the `Sum` term contains the union of all input variables present in a subexpression. Observe that the arity of the union of identical tuples in two (or more) sets is the `Sum` of their arities in the sets being unioned. Thus, `Sum` is effectively a standard sum. Also note that `Sum`'s subexpressions are both commutative and associa-

tive.

`Prod` corresponds to a natural join between two or more subexpressions. As before, observe that the arity of the tuple resulting from the natural join of tuples in two input relations is the product of their arities. Just as in a natural join, output variables present in multiple subexpression schemas are subject to an equijoin – the `Prod` term combines only matching tuples. There are several elements of `Prod` terms, however, that are non-obvious. First, information is passed from left to right – If the schema of an expression contains an input variable which is present in the schema of an expression to its left as an output variable, the input variable is *unified* with the output variable. In effect, the `Prod` term can be thought of as a nested loop, where for every tuple defined by an expression, the expressions to its right are evaluated with some (or none, or all) of their input parameters filled in. Thus, the `Prod` term's schema consists of all output variables defined in its subterms, and those input variables that remain un-unified. Note that the `Prod` term's subexpressions are associative, but only *sometimes* commutative. More on this below. Also note that `Sum` and `Prod` are distributive with each other – `calc_t` effectively describes a ring.

`Neg` does not correspond directly to an operation in relational calculus, but is rather the consequence of an interesting aspect of DBToaster (and [55])'s relational calculus. Specifically, the calculus admits the possibility of negative arities. This sort of anti-tuple also does not correspond to anything in relational calculus, although note that when combined with `Sum`, the result is a set subtraction. `Neg` affects only the arity of its subexpression, its schema is unchanged.

Three more operators are needed to round out the functionality of DBToaster's relational calculus: `Cmp`arisons, `Definition`s and `AggSum`s.

`Cmp` compares two variables or constant values, defining a null relation with a (null) tuple of arity 1 if the comparison holds, or an empty null relation (i.e., the null tuple of arity 0) if the comparison does not hold. `Cmp` terms have no output variables, and zero, one or two input variables. Observe that a comparison may be applied as a filter to an expression (i.e., as an element of a conjunctive WHERE clause) by multiplying the expression by the comparison.

If `Value` terms can be thought of as a way to transfer an output variable's value into the "arity space", then `Definition` terms are the reverse. A `Definition` term extends the schema of its subexpression with a new output variable. The arity of all resultant tuples (i.e., those tuples defined by the subexpression) becomes precisely 1, and the new output variable is in effect assigned to the corresponding arity of the subexpression. There are two elements of interest here: First, note that the new variable is a dependent variable. Second, note that certain forms of `Cmp` terms can be rewritten into `Definition` terms (and visa versa). This is a form of commutativity in `Prod`.

$$m_1[][y] \cdot m_2[][x] \cdot (y = 0) \equiv m_2[][x] \cdot (y \leftarrow x) \cdot m[][y]$$

Finally, `AggSum` terms compute aggregate sums. The arity of the term is the sum of the arities of all tuples defined by the subexpression (observe that only nonzero terms are relevant, as zero terms do not contribute to the sum). If appropriate, an `AggSum` term can be defined with one or more group-by variables. The output schema of the term contains all of the input variables of the subexpression, and only output variables specified in the group-by; all remaining variables are aggregated. Note that this is also the natural way of performing projections in DBToaster's relational calculus – as the value of an expression is

`calc_t` :=			
`Sum`	:	set of `calc_t`	$\{ c_1 + c_2 + \ldots \}$
`Prod`	:	set of `calc_t`	$\{ c_1 \cdot c_2 \cdot \ldots \}$
`Neg`	:	`calc_t`	$\{ -c \}$
`Cmp`	:	`Value`, [<,≤,=,...], `Value`	$\{ v_1 [cmp] v_2 \}$
`AggSum`	:	list of `var_t`, `calc_t`	$\{ \sum_{schema(c)-\{v_i\}} c \}$
`Value`	:	`var_t` or `constant`	
`Relation`	:	`string`, list of `var_t`	$\{ R(v_1, v_2, \ldots) \}$
`External`	:	`string`, list of `var_t`, metadata	
`Definition`	:	`var_t`, `calc_t`	$\{ v \leftarrow c \}$

Figure 5.4: The DBToaster relational calculus

the arity of each tuple, projections are equivalent to sum aggregates.

5.3.1 The DBToaster Workflow

I now describe the workflow that DBToaster uses to translate a database specification into a compiled database engine – whether for use as a standalone process, part of a runtime, or for embedding into an application. This workflow is illustrated in Figure 5.5.

ToaStQL. The input to DBToaster is a database specification – a set of directives written in ToaStQL, a variant of SQL extended to support streaming data. A ToaStQL file consists of two types of statements:

- **Table Definition** statement declare and define an input source, referred to in DBToaster as a *base relation*. A table definition consists of a SQL-like CREATE TABLE statement, followed by a set of options that define how data is inserted into the table. For example, a base relation might be defined by a particular file – when the code generated by DBToaster first

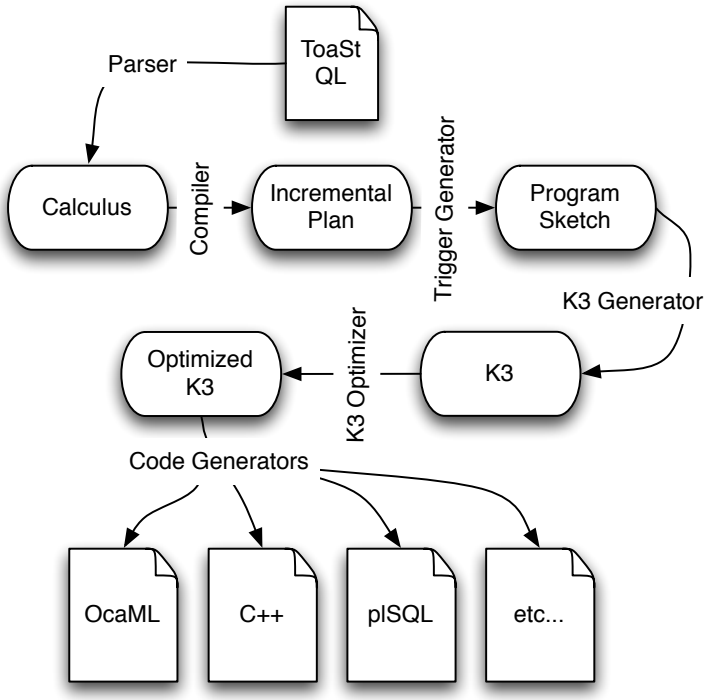


Figure 5.5: The DBToaster compilation workflow

starts up it will "insert" rows into the relation by reading from this file. Data sources can also include sockets, or function invocations performed by the application in which the compiled code is embedded. In addition to the data sources, the table definition statement specifies how to parse data from its data source (analogous to the Hadoop [14] RecordReader interface).

- **Query** statements declare computations using standard SQL SELECT syntax. Depending on how the DBToaster compiler is invoked, the compiler's output will be code defining a data structure to store the query results, or a compiled program that outputs the results of the query, either at the end of the computation or once the end of all input sources is reached.

DBToaster Relational Calculus. The first stage of the DBToaster compiler workflow is to translate ToaStQL into DBToaster relational calculus. Table definition statements are extracted and set aside until the final code-generation phase. Each query statement is parsed and translated into a DBToaster relational calculus (`calc_t`) expression – Code produced by the compiler will maintain results for all queries included in the ToaStQL.

Example 5.3.1 *Consider the following example database specification in ToaStQL:*

```
CREATE TABLE R(A int, B int)
  FROM FILE 'test/data/r.dat' LINE DELIMITED
  CSV (delimiter := ',', schema := 'int,int',
       eventtype := 'insert');
CREATE TABLE S(B int, C int)
  FROM FILE 'test/data/s.dat' LINE DELIMITED
  CSV (delimiter := ',', schema := 'int,int',
       eventtype := 'insert');
CREATE TABLE T(C int, D int)
  FROM FILE 'test/data/t.dat' LINE DELIMITED
  CSV (delimiter := ',', schema := 'int,int',
       eventtype := 'insert');
SELECT sum(A*D) FROM R,S,T WHERE R.B=S.B AND S.C=T.C;
```

This specification includes three input sources and one query. Input sources are specified as tables with three additional fields. The `FROM FILE` *clause indicates that the table is to be read from a file datasource. The* `LINE DELIMITED` *clause indicates the record delimiter, while the* adaptor *clause (*`CSV` *and its parameters in this example)*

clause indicates that each record is stored as two integers delimited by a comma, and that all records found in the file are to be treated as insert events (i.e., the query is to be run as an ordinary database query over the file data).

The query is specified as a simple aggregating `SELECT` *statement, outputting a single value with no group-by terms. The aggregate operator is Sum, thus the root term of the parsed* `calc_t` *expression will be an* `AggSum` *with no group-by variables.*

Nested within the `AggSum` *term is a* `Prod` *of the* `FROM`*, target, and* `WHERE` *clauses of the* `SELECT` *– in that order. In this example, the result is as follows*

```
AggSum([], R[R_A,R_B] * S[S_B,S_C] * T[T_C,T_D] * R_A *
           T_D * (R_B = S_B) * (S_C = T_C))
```

This expression can be further simplified by unifying the two variables `R_B` *and* `S_B` *as follows. First, the equality constraint is rewritten into a definition term. Note that this involves a limited form of commutativity discussed further in Section 5.3.2*

```
AggSum([], R[R_A,R_B] * (S_B <- R_B) * S[S_B,S_C] *
           (T_C <- S_C) * T[T_C,T_D] * R_A * T_D)
```

Finally, the variable in the definition term is replaced within the expression – note that although this changes the schema of the `Prod` *term, the change is possible because the* `AggSum` *aggregates over both schema elements, leaving the entire expression's schema unchanged.*

```
AggSum([], R[R_A,R_B] * S[R_B,S_C] * T[S_C,T_D] *
           R_A * T_D)
```

Incremental Plan. The second and most detailed stage of the compiler workflow is to produce a incremental plan for each query (i.e., each `calc_t` expression). This incremental plan is a tree, branching along two dimensions.

First, the tree encodes the hierarchy of data structures, or *maps* output by DBToaster's recursive IVM compilation process (as overviewed in Section 5.2.1 and implemented in Section 5.3.3). Each map corresponds to a specific `calc_t` expression, and the data structure responsble for incrementally maintaining it. Thus, the incremental plan is similar in function, if not form, to the role of a query plan in a normal relational database engine.

This hierarchy is constructed out of two alternating layers: (1) Each map is maintained by one or more *invocations* – `calc_t` expressions evaluated at runtime. Each invocation is invoked in response to an *event* – the insertion or deletion of a tuple from one of the base relations. (2) Each invocation is *materialized* with respect to zero or more maps – portions of the invocation materialized into a map are incrementally maintained, resulting in less work at runtime.

The second "dimension" of the incremental plan is the range of possible materialization decisions made by the compiler. As part of the compilation process a different ways of implementing an invocation in terms of maps or base relations are identified. This includes not only decisions regarding what portions of the invocation to include, but also the type of data structure used – A map implemented as a hash table is most efficient for simple key lookups, while one implemented as a range tree can implement inequality comparisons more efficiently. This decision is discussed further in Section 5.4.

It should be noted that – as will become evident, making materialization de-

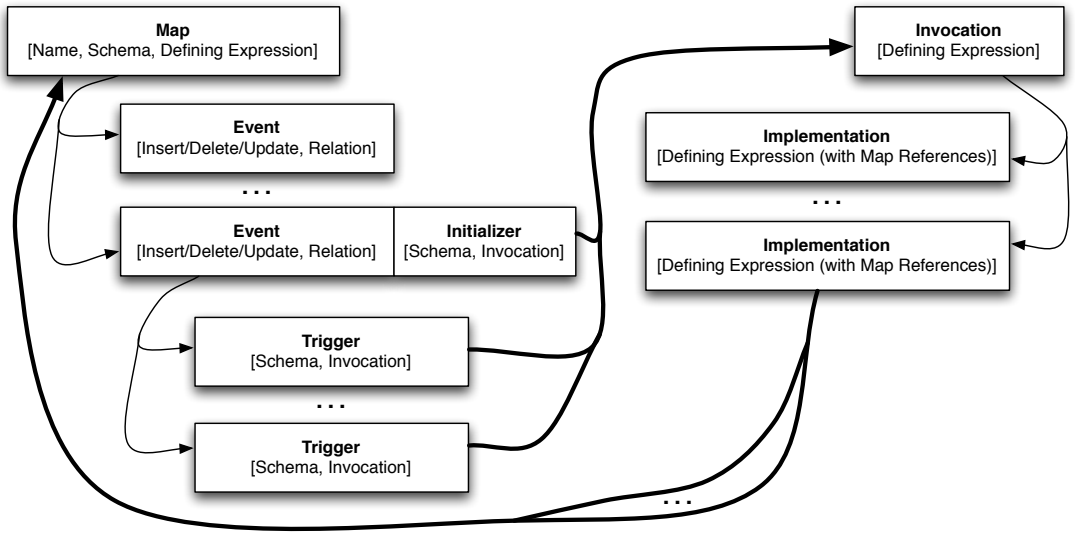


Figure 5.6: The two-layer recursive structure of a DBToaster incremental plan.

cisions over *polynomial* `calc_t` expressions (expressions containing `Sum` terms) is nontrivial. Hence, at this stage, the invocation will be first rewritten into a set of monomials (i.e., a `Prod` of non-`Sum`s). Although these monomials are factorized back into polynomial form, it is possible for various expression simplifications applied at this stage to change the expression's schema – consequently, maps in the incremental plan can include several invocations for a single event.

The incremental plan data structure is presented in Figure 5.3.1. Note that the incremental plan is an extension to the basic `calc_t` framework – maps, invocations, and implementations are specified in terms of a defining `calc_t` expression. The process of compiling a incremental plan is described in further detail in Section 5.3.3.

The resulting incremental plan is thus not just a single strategy, but a range of different possible implementations. In general, only a single strategy will be

implemented into compiled code – this selection task is performed by a *materializing optimizer*, applied to the incremental plan after it is generated by the compiler. In addition to selecting between the different options presented to it by the compiler, the materializing optimizer eliminates duplicate maps (in effect, turning the tree into a DAG). Finally, the materializing optimizer can collapse multiple layers of the incremental plan (now DAG), resulting in additional computation work when evaluating an invocation, but reducing the amount of storage required – user preferences decide the storage/computation tradeoff.

Note that the materializing optimizer extends the incremental plan, rather than pruning it. The output of the optimizer is not just a single default materialization, but a weighted set of options. In addition to providing users with valuable feedback, this will enable future implementations of DBToaster to restructure the engine's implementation at runtime if it should be beneficial to do so.

A final aspect of the materializing optimizer that deserves discussion is the decision to partially materialize a map: A map with input variables in its schema can generally not be materialized in its entirety – Either the input variables must be removed (via unification, or by moving them into the invocation), or the map can be instantiated as a *caching map*. In this case, the map's values are maintained for a certain subset of known input variables. When the map is accessed with previously unseen input variables, the map is *initialized* for these new input variables, and incrementally maintained henceforth. Note that this initialization process requires the execution of an invocation.

The materializing optimizer can take this idea still further. It can choose to instantiate the map only for some values of the output variables as well –

for example only those variables that match a constraint defined by the output variables. If it chooses this approach, then, as in the case of variables, the code produced by DBToaster must first verify whether a map has been initialized for the particular output variables whenever the map is read from.

Program Sketch. From the default materialization produced by the materializing optimizer, DBToaster produces a *program sketch*. This is a rough outline of the code to be produced, specified as a set of input sources (taken from the ToaStQL), a set of maps to materialize, and a compilation of invocations to execute for each event in order to maintain and initialize the materialized maps.

The program sketch's events are computed by recursively traversing the default materialization. During this traversal, all unique maps are recorded (recall that the default materialization is a DAG). Furthermore, event invocations are recorded and sorted by event type.

K3. The next stage of compilation is to implement the program sketch by translating every invocation in every event in the sketch into a set-based functional language called Kleisly K2, based on [17] (K3). The result is one functional program for each event:

- `AggSum` terms are translated into a function that computes the aggregate of the set defined by the term's subexpression.
- `Prod` terms are translated into the application of mapping functions: two `calc_t` terms multiplied together results in a translation of the left-hand-side term. If the right-hand side term has no output variables (i.e., a `Value`, `Cmp`, no-group-by `AggSum`, or a `Neg` or `Definition` of one of the above) in its schema (or only dependent variables, in the case of a

`Definition` term), the generated mapping function multiplies the arities of the left and right hand sides, and if appropriate extends the schema of the output set. Note that this sort of term is referred to as a *singleton*. If the right hand side is not a singleton, the product is translated into a nested loop join.

- `Sum` terms are translated into set addition.
- `Relation` and `External` terms are translated into set reads. Note that when the read occurs, some variables may be defined; reads are performed by *slicing* – extracting all values of the map or relation that match a specific, equality-only selection predicate.
- All remaining terms are translated into mapping functions as described above in the translation of `Prod`.
- In addition to the terms of the `calc_t` invocations being compiled, K3 also encodes several other elements of the program sketch: Most noticeably, all accesses (both read and write) to maps may require initialization of the map, if the materializing optimizer has chosen not to fully materialize the map (as discussed above). K3 explicitly encodes this check as an if statement prior to reading the map.

The use of a functional representation occurs for two reasons. First, K3 acts as an intermediate representation to facilitate translation between the extremely abstract DBToaster relational calculus, and the much more concrete DBToaster's output languages (C++, OCaml, etc...). Second, and more importantly, K3 is extremely amenable to various functional, inter-query, and order-of-operations optimizations that are either difficult or outright impossible to

express in DBToaster relational calculus. These optimizations are discussed further in Section 5.3.4

Compiled Code. Finally, the optimized K3 is translated into a more commonly used programming language – either OCaml, C++, or Postgres plSQL. The translation to OCaml is remarkably straightforward – both are functional languages. When translating into C++, plSQL, or any other imperative language, the K3 code is first translated into an imperative intermediate representation – unlike K3, where all operations are performed over sets, the intermediate representation explicitly loops over set elements.

An example of the stages of compilation is illustrated in Appendix B.

5.3.2 Simplifying Calculus Expressions

During various stages of the compilation process, DBToaster performs several simplifications on the `calc_t` expressions it is working with – This produces a less complex expression that is semantically equivalent, but in a normalized form that is easier to work with.

Unification. The input and output variables (both input and output, as well as their types) of a calculus expression can always be computed from the expression; external metadata is not required.

Variable unification has an impact on the schema of an expression. Not only can it potentially remove input and output variables, but it can turn input variables into output variables (i.e., if an input variable is unified with an output

variable) [1].

Variable unification itself takes two forms: (1) downward, and (2) upward.

Upward unification is the harder of the two, because it requires knowledge of input variables. A comparison between two variables $x = y$ can be unified upwards using one of the following two rules:

- $e := m[x][] * (x = y)$ becomes $(x \leftarrow y) * m[x][]$ iff x is not bound anywhere to the left of e (i.e., an output variable).
- $e := m[y][] * (x = y)$ becomes $(y \leftarrow x) * m[y][]$ iff y is not bound anywhere to the left of e.

It is possible for a variable to be bound at multiple points in an expression (i.e., it appears in multiple relation terms). The upward unification process requires that the newly bound variable be lifted above all of the variable's bind-points.

Downward unification is simpler. An expression of the form:

$$\texttt{AggSum}([dom(m[\ldots][x,\ldots]) - \{x\}], (x \leftarrow y) * m[\ldots][x,\ldots])$$

is equivalent to $m[\ldots][y,\ldots]$. Interestingly, note that this process is somewhat reversible. Given a variable x with a name not defined in the environment or domain of expression e (and just to be safe, not used in the expression either), $m'[\ldots][\ldots]$ into

$$\texttt{AggSum}([m'[\ldots][\ldots]], (x \leftarrow \{anything\}) * m'[\ldots][\ldots])$$

[1] Of course, this poses some limitations with respect to extracted expressions. The schema of a map can not be changed after the fact. Even if it is possible to perform further unification on a delta expression, the schema used to access the map must be further adjusted to match.

Here, $\{anything\}$ can in fact be replaced by anything with an empty schema.

In particular, this allows factorization over variable definitions. For example:

$$\texttt{AggSum}([dom(m[\ldots][x,\ldots]) - x], (x \leftarrow y) * m[\ldots][x,\ldots]) + m'[\ldots][\ldots]$$

$$\texttt{AggSum}([dom(m[\ldots][x,\ldots]) - x], (x \leftarrow y) * m[\ldots][x,\ldots])$$
$$+\texttt{AggSum}([dom(m'[\ldots][\ldots])], (x \leftarrow y) * m'[\ldots][\ldots])$$

$$\texttt{AggSum}([dom(m[\ldots][x,\ldots]) - x], ((x \leftarrow y) * m[\ldots][x,\ldots]) + ((x \leftarrow y) * m'[\ldots][\ldots]))$$

$$\texttt{AggSum}([dom(m[\ldots][x,\ldots]) - x], (x \leftarrow y) * (m[\ldots][x,\ldots] + m'[\ldots][\ldots]))$$

In other words, definition terms can be lifted as far up through summation terms as we like, as long as there are no variable naming conflicts in the terms being distributed across. Once a definition term is at the very top, we can perform downward unification as desired.

Monomialization. In general, it is easier to work with expressions containing no `Sum` terms. The materialization partition process performed by `compile_invocation` (described below) is a perfect example of this. If one thinks of the entire `calc_t` expression as a polynomial (i.e., a sum of products), then sumless `calc_t` expressions are effectively monomials. Monomialization is performed by lifting `Sum`s to the top of the expression by applying the distributive law. The set of monomial terms can be expressed with an implicit `Sum` as a list of monomial `calc_t` expressions – The compiler algorithms described below in Section 5.3.3 use this representation.

Term Lifting and Merging. Two final simplifications are applied: (1) `Neg` terms are pushed down through the product terms so that each `Neg` wraps a singleton, `External`, or `Relation` term, and (2) `AggSum` terms are pulled up to the top of each monomial or `Definition`, and immediately adjacent nested `AggSum`s are

merged by dropping the inner aggregate. Similarly, immediately nested `Prod` terms are merged, as are `Sum`s.

The result of these rewritings is a: (1) `Sum` of (2) `AggSum`s of (3) `Prod`s of (4) optional `Neg` of (5) everything else. The subexpression of each `Definition` term is also rewritten into a similar hierarchy.

5.3.3 Compiling Calculus Expressions

The core of the DBToaster compilation process is the construction of a incremental plan from a `calc_t` expression. Note that a incremental plan can have either a map or an invocation as its root – In the former case, the query is completely materialized and incrementally maintained, while in the latter case, portions of the query may be computed at lookup time (i.e., when the code into which DBToaster's output is nested tries to read query results). As a consequence, DBToaster's compilation process is constructed as two mutually recursive steps, either of which can act as an entry point. The compilation process begins by invoking either `compile_invocation` or `compile_map`.

Compiling Invocations. The `compile_invocation` function, described in simplified form in Algorithm 10 translates a `calc_t` expression into a incremental plan rooted at an invocation node. The primary purpose of this function is to identify a set of different possible materializations of the input expression – only one heuristic is illustrated in Algorithm 10.

Two challenges present themselves at this stage:

First, different portions of the invocation's schema may be defined by inde-

pendent portions of the invocation's defining expression – It may be possible to rewrite the expression as a cartesian cross product. Materializing the entire expression as a map results in wasted resources, both in maintaining the map, as well as the storage requirements of keeping the entire cross product around. `compile_invocation` identifies such independencies by first simplifying the expression into a set of monomials (as described in Section 5.3.2). For each monomial, independent subexpressions are identified by performing a union-merge over the schemas of each term in the monomial. Once the independencies have been identified, each independent subexpression is collected for materialization as a separate map.

Input variables appear in an expression exclusively due to the presence of a `Cmp` or `Value` term. The second challenge of compiling invocations is to decide whether or not to create a map with input variables in its schema – these terms can be included in the map's definition or evaluated when the invocation is evaluated. This materialization decision is dependent on the way in which the expression will be evaluated – If the domain of values that the input variable(s) take is small, incorporating input variables into the map will allow the expression to be incrementally maintained in its entirety. However, if the domain is large, multiple different copies of the data will need to be incrementally maintained, and it will become more efficient to evaluate the expression as part of the invocation. DBToaster allows the user to decide between these two materialization decisions.

Compiling Maps. The `compile_map` function, described explicitly in Algorithm 11 translates a calc_t expression into a incremental plan rooted at a map node. The primary purpose of this function is to enumerate all deltas of the

$$
\begin{aligned}
\delta_{R(\vec{x})}(f+g) &\rightarrow (\delta_{R(\vec{x})}f) + (\delta_{R(\vec{x})}g) && (5.1)\\
\delta_{R(\vec{x})}(f*g) &\rightarrow ((\delta_{R(\vec{x})}f)*g) + (f*(\delta_{R(\vec{x})}g)) + ((\delta_{R(\vec{x})}f)*(\delta_{R(\vec{x})}g)) && (5.2)\\
\delta_{R(\vec{x})}(-f) &\rightarrow -(\delta_{R(\vec{x})}f) && (5.3)\\
\delta_{R(\vec{x})}(v_1 \phi v_2) &\rightarrow 0 && (5.4)\\
\delta_{R(\vec{x})}\texttt{AggSum}([\vec{y}], f) &\rightarrow \texttt{AggSum}([\vec{y}], (\delta_{R(\vec{x})}f)) && (5.5)\\
\delta_{R(\vec{x})}R(\vec{y}) &\rightarrow \prod_i (y_i \leftarrow x_i) && (5.6)\\
\delta_{R(\vec{x})}R'(\vec{y})\{R \neq R'\} &\rightarrow 0 && (5.7)\\
\delta_{R(\vec{x})}(v \leftarrow f) &\rightarrow (v \leftarrow (\mathbf{f} + \delta_{R(\vec{x})}f)) - (v \leftarrow \mathbf{f}) && (5.8)
\end{aligned}
$$

Figure 5.7: The DBToaster relational calculus delta (δ) rewrite rules (based on [55]). ϕ is any comparison operator ($=, <, >, \neq, \leq, \geq$). 0 is the empty null schema relation.

expression, and then to invoke `compute_delta` (described in Algorithm 12) to compute it. All events affecting the map are identified and their effect on the map (in terms of inocation) are stored. Delta computations are relatively straightforward and drawn virtually directly from [55]. These rules are restated in Figure 5.7.

Recall that the delta of an expression in DBToaster relational calculus is strictly simpler – that is, the number of Relation terms appearing in it is reduced (by one). Note however, that Rule 5.8 does not produce a strictly simpler expression. Unlike all other terms in DBToaster relational calculus, the delta of a `Definition` term is not simpler – the delta contains the original term as well as the delta. Fortunately, the delta term can be simplified by extracting its subexpression and incrementally maintaining it as a new *supplemental map*. This may result in runtime evaluation of the subexpression if the subexpression contains an input parameter.

A challenge faced by `compile_map` itself is the complexity of the updates be-

ing created. In order to identify independent subexpressions, `compile_invocation` must first produce a set of monomials – Though this simplifies the task of identifying independent subexpressions, it can create a large number of independent sum terms. As part of the process of `compile_map` it is desirable to factorize these polynomial expressions back together as far as possible.

A simple instantiation of the factorization process is shown in Algorithm 13. In this greedy implementation, the goal is to merge terms as extensively as possible – The algorithm identifies the most common term in the monomials that it is presented with, and merges together those monomials containing the term. The algorithm then recurs separately on the set of monomials containing the term and the set not containing the term.

Factorize uses an intermediate `factor_tree` representation of each monomial:

```
factor_tree = (term:calc_t) * (children:factor_tree list)
```

Every node in this tree represents a single calc_t. A forest of such trees represents a sum of terms (a polynomial). Thus, the calc_t defined by the node is the product of the `term` and the (parenthesized) sum of all of the children.

5.3.4 Optimizing K3

The K3 optimizer repeatedly performs rewrites a K3 expression in the ways described below, until a fixed point is reached. These optimizations are based on those in [17].

Function Composition. Functions that are applied consecutively to the output

Algorithm 10: compile_invocation(*definition*, *schema*)

Require: calc_t *definition*, schema_t *schema*

Ensure: calc_t *implementation*, list of map *references*

1: $monomials \leftarrow$ `simplify`$(definition)$; $impl \leftarrow$ `Const(0)`
2: **for all** $m\_term \in monomials$ **do**
3: $factors \leftarrow$ **match** m_term **with** `Prod`$(factors)$
4: $groups \leftarrow \emptyset$; $consts \leftarrow$ `Const(1)`; $comparisons \leftarrow$ `Const(1)`
5: **for all** $factor \in factors$ **do**
6: Find a set $\{g_i\} \subseteq groups$ s.t. $\{g_i\}$ contains all elements of $groups$ that produce an output variable $v \notin schema$ that appears (as an input or output variable) in $factor$.
7: **if** $|\{g_i\}| = 0$ **then**
8: **if** $(factor$ **matches** $(i \leftarrow f))$ **and** $(f$ **includes** `Rel`$(*))$ **then**
9: $consts \leftarrow consts * (i \leftarrow$ `compile_map`$(f))$
10: **else if** $factor$ **matches** $(i \leftarrow f)$ **or** `Value`$(*)$ **or** `Cmp`$(*,*,*)$ **then**
11: $consts \leftarrow consts * factor$ **else** $groups \leftarrow groups@factor$
12: **else**
13: **if** $(factor$ **matches** `Cmp`$(\mathtt{v}_1, *, \mathtt{v}_2))$ **and** $(v_1$ **or** v_2 **matches** `Value(Const`$(*))$ **or** `Value(Var(k))` s.t. $k \in schema)$ **then**
14: $comparisons \leftarrow comparisons * factor$
15: **else**
16: $groups \leftarrow (groups - \{g_i\})@(g_0 * \ldots * g_n * factor)$
17: $impl \leftarrow impl + \left(consts * \left(\prod_{g_i \in groups}$ `compile_map`$(g_i)\right) * comparisons\right)$

Algorithm 11: compile_map(*definition*, *schema*)

Require: calc_t *definition*, schema_t *schema*

Ensure: map *map*, map list *supplements*

1: **for all** $R \in$ relations appearing in *definition* **do**

2: $(map.deltas[R], supplements_i) \leftarrow$ `compute_delta`$(R, definition)$

3: $map.deltas[R] \leftarrow$ `factorize`$(map.deltas[R])$

4: **return** $(map,$ **concat**$(supplements_i))$

of the prior function are composed into a single operation.

Function composition is equivalent to adjusting the parenthesization of the calculus expression. The advantage to implementing it in the K3 optimizer is that it simplifies the translation from calculus to a language closer (slightly at least) to the bare metal.

A similar effect could be achieved by establishing certain design patterns (e.g., $(a < 0) * (b < 0)$ is translated into a filter of the conjunction of the two terms, rather than a two-stage filter), but the simpler way reduces bugs and has a significant impact when combined with the remaining rewrite rules.

Function Inlining. If the optimizer can assert that a function is defined and applied only to a single code block, the function application is replaced by an inlined version of the function (i.e., a copy of the function definition with its parameter variable(s) replaced by the parameter values used to invoke the function.

This step could theoretically be avoided; Its primary purpose is to enable a more intuitive translation from Calculus to K3. Rather than establishing all

Algorithm 12: compute_delta(R, *term*)

Require: relation $R(var_i)$, calc_t *term*

Ensure: calc_t *delta_term*, map list *supplements*

1: **match** *term* **with**

2: | `Sum`$[x_i] \rightarrow Sum$[`compute_delta`(x_i)]

3: | `Prod`$[x_0, x_i] \rightarrow$

$(d_0, supp_0) \leftarrow$ `compute_delta`(x_0)

$(d_{rest}, supp_{rest}) \leftarrow$ `compute_delta`(`Prod`$[x_i]$)

$delta\_term \leftarrow x_0 * d_{rest} + d_0 * \texttt{Prod}[x_i] + d_0 * d_{rest}$

$supplements \leftarrow supp_0 @ supp_{rest}$

4: | `Neg`$(x) \rightarrow$ `Neg`(`compute_delta`(x))

5: | `Cmp`$(\_, \_, \_) \rightarrow (delta\_term, supplements) \leftarrow (0, \emptyset)$

6: | `AggSum`$(x) \rightarrow$ `AggSum`(`compute_delta`(x))

7: | `Rel`$(name, termvar_i) \rightarrow$

$supplements \leftarrow \emptyset$

$delta\_term \leftarrow$ **if** $name = R$ **then** $\prod_i (termvar_i \leftarrow var_i)$ **else** 0

8: | `External`$(\_) \rightarrow$ **fail**

9: | `Definition`$(v \leftarrow x) \rightarrow$

$(x', supplements) \leftarrow$ `compile_map`(x)

$delta\_term \leftarrow ((v \leftarrow (x' +$ `compute_delta`$(\texttt{x}))) - (v \leftarrow x'))$

Algorithm 13: factorize(*monomials*)

Require: A calc_t (as above) list *monomials*

Ensure: A factor_tree list $forest$, which can be converted into a single polynomial with the `merge_forest` function defined in Algorithm 14

1: $count_* \leftarrow 0$ {Greedy heuristic: find the most common term}
2: **for all** $m \in monomials$ **do**
3: **for all** unique $term \in m$ **do**
4: $count_{term} \leftarrow count_{term} + 1$
5: $common\_term \leftarrow \texttt{argmax}_{term}(count_{term})$
6: **if** $count_{common\_term} \leq 1$ **then**
7: $forest \leftarrow \emptyset$ {No commonality between monomials – return what we have}
8: **for all** $m \in monomials$ **do**
9: $forest \leftarrow forest \cap \{(\texttt{term} : m, \texttt{children} : \emptyset)\}$
10: **else**
11: $in\_terms \leftarrow \emptyset$; $out\_terms \leftarrow \emptyset$ {Split the monomials depending on whether or not they contain the common term}
12: **for all** $m \in monomial$ **do**
13: **if** $common\_term \in m$ **then** $in\_terms \leftarrow m$ **else** $out\_terms \leftarrow \frac{m}{common\_term}$ **end if**
14: $new\_tree \leftarrow (\texttt{term} : common\_term, \texttt{children} : \texttt{factorize}(in\_terms))$
15: $forest \leftarrow \texttt{factorize}(out\_terms) \cap \{new\_tree\}$

Algorithm 14: merge_forest($forest$)

Require: A factor_tree list $forest = term_i, children_i$

Ensure: *polynomial*, the calc_t representation of $forest$

$polynomial \leftarrow \sum_i term_i * \texttt{merge\_forest}(children_i)$

of the bindings from input to output variables by hand, DBToaster generates a large number of functions (one per product term) and then relies on the optimizer to simplify the generated expression down as far as possible.

If-Lifting. If conditions appearing in the K3 code (i.e., those created by a check for whether a map is initialized for the values being read) are lifted as high in the K3 syntax tree as possible. Operations (i.e. Apply, Flatten, Function definition, etc...) with an IfThenElse node as a child switch places with the IfThenElse node. The operation is applied to both the IfThenElse node's Then and the Else children, and the IfThenElse node becomes the new parent.

This swap is not always possible. In particular, an IfThenElse can not be lifted so far that a variable in its If clause goes out of scope. The IfThenElse node is lifted up to the highest possible point in the syntax tree. Having the conditional this far up (combined with the Inlining and Composition optimizations) ensures that if statements do not occur in deeply nested loops.

IfThenElse nodes appear exclusively as the result of map accesses with non-zero initializers. Thus, this optimization has two roles: (1) a parenthesization optimization akin to what Function Composition does, and (2) un-nesting multiple levels of initializers.

The parenthesization optimization is the primary effect of if-lifting. In particular, this occurs when dealing with the cross-product of two maps. For example:

$$m_1[x][y] * m_2[z][]$$

In this expression, both m_1 and m_2 will need to be initialized, but without some messy factorization code, the initializer for m_2 will be tested for once for each value of b in m_1. If lifting allows the test to be performed only once, effectively

factorizing the expression.

Un-nesting multiple levels of initializers is of use, as the inner initializers are likely to be simpler. If the inner initializer depends only on variables declared outside of the outer initializer (and fewer variables than the outer initializer as well) it may be possible to lift its initializer further up than the outer initializer, again reducing the if-clauses in an inner loop. Note however, that in general DBToaster's materialization heuristics do not produce initializers dependent on maps that may themselves need to be initialized.

Block-Lifting.

Like OCaml, K3 includes an imperative-style semicolon operator for defining code blocks. The K3 semicolon operator accepts a left-hand (terminal) side operation that evaluates to a non-value (equivalent to OCaml's unit type), and a right-hand side operation that defines the operator's output. The terminal operation is invoked first, and produces only side effects (e.g., by updating map values).

Like if statements, terminal operations are lifted as far up in the syntax-tree as possible – As with ifs, it is desirable for this statement to be evaluated as far up in the loop structure as possible. A terminal expression can not be lifted out of the scope of a variable it uses, and any if statements that it is lifted past must be split. For example:

$$\texttt{If } A \texttt{ Then } f(x); B \texttt{ Else } C \rightarrow (\texttt{If } A \texttt{ Then } f(x) \texttt{ Else } UNIT); (\texttt{If } A \texttt{ Then } B \texttt{ Else } C)$$

If-Factorization. If statements or conditions are eliminated if they can be proven to be ineffectual. This occurs if the then and else clauses can be shown to be

equivalent, or if the condition can be shown as always being true or false.

Block-Factorization.

Two if statements with identical conditions, both defined as part of a single block are merged together. Similarly, functions being applied to the same value are merged into one function with a single block inside. For example:

$$\texttt{Apply}(\lambda x.f(x), A); \texttt{Apply}(\lambda x.g(x), A) \rightarrow \texttt{Apply}(\lambda x.(f(x); g(x)), A)$$

K3 Optimizations in DBToaster Relational Calculus. It is possible to view many of the optimizations used in this stage as simply different "parenthesizations" of a `calc_t` expression. For example, composing two filter operations can be thought of as the associative equivalence:

$$(m[][x, y] \cdot (x > 0)) \cdot (y > 0) = m[][x, y] \cdot ((x > 0) \cdot (y > 0))$$

However, two things prevent this approach from being more effective: (1) Using associativity as a way of expressing these sorts of optimization decisions necessitates an extremely complex infrastructure for generating compiled code – In general, the product operator can be translated into anything from function application to a natural join. Performing these sorts of optimizations on a functional language makes it possible to have two "dumb" translation layers (`calc_t` to K3 and K3 to output language) and an independent and isolated optimizer, rather than a single translation layer that is intricately linked into the optimization process. (2) DBToaster relational calculus does not have a mechanism for expressing cross-expression (i.e, block-level) optimizations – each `calc_t` represents only a single computation. The K3 optimizer can combine multiple independent loops.

5.4 Managing Storage in DBToaster

Typical DDMS workloads require large state, necessitating the use of suitable storage techniques. Compared to traditional DBMS with primarily ad-hoc query workloads, DDMS have more information to use when constructing a task-specific storage solution.

With respect to its storage layer, DBToaster's strategy for obtaining the performance characteristics required of a DDMS is twofold:

1. The DBToaster compiler produces data structures designed specifically for the compiled DDMS' target query workload.
2. By analyzing the patterns with which data is accessed, DBToaster constructs a data layout strategy (for pages on a disk, servers in a cluster, etc.) that limits IO overhead.

5.4.1 Data Structures

The simplest abstraction used by DBToaster to represent materialized views is the multi-key (i.e., multi-dimensional) map. Fundamentally, a multi-key map is a simple key-value store with structured (i.e., schema-defined) keys and values, as well as some iteration capabilities.

Though similar to relational tables in this respect, there are two subtle, but critical differences:

- The map's value at each point is defined not by the data stored in it, but

rather as a function (subquery) over the state of the database.

- Unlike a relational table, where absent keys imply NULL values, multi-key maps are defined for all keys conforming to the map's schema.

These two differences are closely related. A map's value must be defined for all keys, because all updates are specified as deltas. In the absence of prior state, this value can be derived from lower level maps, or the base relations.

Interestingly, for non-nested queries without inequalities, this value always begins at zero. So long as a map is maintained incrementally in its entirety, DBToaster never needs to compute the initial value.

Slicing. Generated code does not typically iterate over the entire map data-structure. In the common case, statements iterate over keys matching a selection predicate. Consequently code generated by DBToaster performs map lookups by slicing – selecting lower-dimensional cuts through the map's key space. The code provides a partial key, fixing some dimensions and establishing (implicitly) an iterator over the remaining ones.

Example 5.4.1 *Recall the code listing in Figure 5.3, and in particular the second statement of the* `on_insert_customer` *trigger.*

```
m[][ordkey, sprior] += m_c[][ck, ordkey, sprior];
```

This statement is a partial lookup on map m_c with only `ck` *defined by the trigger arguments. The resulting value (which in turn, is used to update m) is equivalent to a 2-key map – the slice taken by fixing* `ck` *to the corresponding trigger argument value.*

In addition to exact and partial lookups, DBToaster maps support range lookups by inequality predicates.

Implementing Maps. In their simplest form, out-of-core maps are implemented by a simple relational-style key-value store with secondary indices [69].

Inequality predicates, and aggregations including such predicates, are implemented efficiently using maps that store cumulative sums [41]. Maps can apply compression techniques to address frequently repeating data.

DBToaster customizes the data structures backing each materialized view based on statement-level information on accesses, applying static compile-time techniques to construct specialized data structures.

With substantial specialization of data structures as part of compiling transitions, DBToaster is free to consider a range of runtime issues in data structure tuning and adaptation, including how to best perform fine-grained operations such as incremental and partial indexing [82]. The key challenge to be addressed is how to provide data structures with a low practical update cost (avoiding expensive index rebalancing and hash-table re-bucketing) while gradually ensuring the lookup requirements of DBToaster's data structures are retained over time, amortizing data structure construction with continuous query execution.

5.4.2 Partitioning and Co-clustering

Database partitioning and co-clustering decisions are traditionally made based on a combination of (a) workload statistics, (b) information on schema and integrity constraints (such as key-foreign key constraints, a popular basis for co-

clustering decisions), and (c) a body of expert insights into how databases execute queries. Ideally, such decisions should be based on a combination of (a), (b), and a *data flow analysis* of the system's query execution code, instantiated with the query plan, or view maintenance code. In classical DBMS however, this is too difficult to be practical.

Fortunately, data flow analysis turns out to be feasible for compiled DDMS transition programs: in fact, it is rather easy. A transition program statement reads from several maps and writes to one, prescribing dependencies between those maps occurring on the right-hand side of the statement (reading), and the one on the left-hand side (writing). As pointed out in Section 5.2, transition program statements admit a perfectly data-parallel implementation: consequently, a statement never imposes a dependency between two items of the same map and any horizontal partitioning across the involved maps map keeps updates strictly local.

Using these data flow dependencies, partitioning and co-clustering decisions can be made by assigning a data statistics-dependent cost to placing dependent data in different partitions and solving a straightforward min-cut style optimization problem.

DBToaster implements its analysis of data-flow dependencies by abstracting these dependencies into a *messaging graph*, such as the one illustrated in Figure 5.8. A transition function is represented as a bipartite directed hypergraph; nodes on the left represent portions of the database being read from, nodes on the right represent a portions being written to, and each hyperedge represents an independent subtask of the transition function. An example is shown in Figure 5.8a.

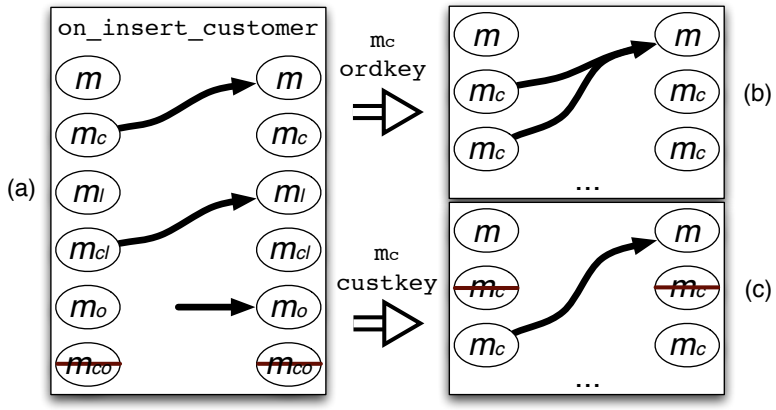


Figure 5.8: An example of a messaging graph based on Example 5.2.1's query q (a) The messaging graph for the `on_insert_customer` event. (b) The effects of splitting view m_c on `ordkey`. (c) The effects of splitting m_c on `custkey`.

For example, consider the transition function that results from an update to the customer table. One specific subtask of this transition reads m_c and writes to m. Treating each view as a node, this task has one edge with one read node and one write node.

DBToaster considers database layout in terms of how it partitions data across a physical medium (i.e., memory, disk pages, or a cluster). Viewed through the messaging graph, a partitioning is an assignment of all nodes in the graph to one (or more, in the case of replication) partition. For example, if they were small enough, m and m_c might be placed on one disk page each. Thus, the subtask requires a read from one page, and an update to a second page – 2 pages in total.

Subdivision of individual views is represented in the messaging graph by splitting graph nodes. Of particular interest is how the new nodes interact with the hyperedge(s) connected to the original node. As the split occurs, a node may stay connected to a hyperedge, the hyperedge may also be split, or in some

cases, only one node will remain connected (see Figure 5.8b,c). DBToaster can exploit the limited range of node split/hyperedge interactions to select an effective partitioning scheme. This distinction is very similar to the different composition operators of Dryad's [44] vertex programs – DBToaster can easily assign the correct composition operator to each vertex in a messaging graph.

Example 5.4.2 *When partitioning m_c, horizontal partitioning on the value of* `ordkey` *increases the number of nodes connected to the* `on_insert_customer` *task edge, while using* `custkey` *does not provoke an increase. If the data represented by these nodes is split across multiple disks or servers, the computation must still access all of them. The roles are reversed for the* `on_insert_order` *task edge. Under the (minimal) assumption that both types of insert events occur with identical frequency, DBToaster partitions on both keys to minimize the number of connected nodes. If additional information is available about the distribution of event frequencies, DBToaster can partition accordingly.*

Additional knowledge about the dataset enhances the messaging graph produced by DBToaster. For instance, the ER diagram can be integrated into the messaging graph – the chain of foreign key dependencies in q is strictly hierarchical. DBToaster uses this information and creates partitions along a single axis with a secondary index to bound the number of partitions accessed by each update subtask, with respect to the total number of partitions generated. Similarly, this information is used by DBToaster to select a partitioning scheme that places nodes typically connected by a subtask into a single partition; this is analogous to co-clustering in a traditional DBMS.

5.5 DBToaster in the Cloud

Scaling up a DDMS requires not only storing progressively more data, but also a dramatic increase in computing resources. As alluded to in Section 5.4, DDMS and their corresponding transition programs are amenable to having their data distributed across a cluster:

1. The only data structures used by transition programs are maps, which are amenable to horizontal partitioning.
2. At the granularity of a single update, iterative computations are completely data-parallel.
3. The effect of a sequence of updates (i.e., executing the corresponding trigger functions) is independent of the order in which the updates are applied.

Update Processing Consistency and Isolation. In DBToaster, transition functions are created for serial execution – the code of a transition function assumes that it is operating on a *consistent* snapshot of the DDMS's state. The entire sequence of statements composing the trigger function must be executed atomically as one operation, to ensure that each statement operates on maps resulting from fully processing the update stream prior to the update that fired the trigger. Thus the effects of updates should be fully isolated from each other. Similar issues and requirements have been raised before in the single-site context of view maintenance with the "state bug" [22].

Our requirement of processing updates in such an order is conservative, indeed DBToaster could apply standard serializable order concurrency control

here to simultaneously process updates that do not interact with each other. Ensuring atomicity is the first of the two core challenges involved in constructing a distributed DDMS runtime.

The underlying goal is to develop simple techniques that avoid heavyweight locking and synchronization of entries in massively horizontally partitioned maps. This goal is achieved by being conservative, and focusing on lightweight protocols.

5.5.1 Distributed Execution

Each update in DBToaster's distributed DDMS runtime design employs three classes of actor:

- *source nodes*: Nodes hosting maps read by the update's trigger function (maps appearing on the right-hand side of the function's statements).
- *computation nodes*: The nodes where statements are evaluated.
- *destination nodes*: Nodes hosting maps written to by the update's trigger function (maps appearing on the left-hand side of the function's statements).

Note that these actors are logical entities; it is not necessary (and in fact, typically detrimental) for the actors to be on separate physical nodes within the cluster. Note also that the other extreme is not typically possible; distribution always introduces some amount of separation; the source nodes for one update will be the destination nodes for another, and visa versa.

Introducing a distinction between the different tasks involved in update processing makes it possible to better understand the tradeoffs involved in the second core challenge: selecting an effective partitioning scheme that intelligently determines placements of logical entities in order to best utilize plentiful hardware to handle a large update stream and DDMS state.

5.5.2 Execution Models

The issue of atomicity is addressed in DBToaster with two different execution models:

- A protocol that provides a serial execution environment for transition programs.
- An eventual consistency protocol that provides the illusion of serial execution.

Serial Execution. The most straightforward way of achieving atomicity is to ensure that trigger functions are evaluated serially. However, requiring all nodes in the cluster to block on a barrier after every update is not scalable.

A similar effect can be achieved more efficiently by using fine-grained barriers, where each update is processed by first notifying all of the update's destination nodes of an impending write. Reads at the update's source nodes are blocked while writes from prior updates are pending.

Serial execution requires a global ordering of updates as they arrive at the DDMS. Techniques to achieve this include:

- Updates arrive only from a single producer (e.g., the cluster is maintaining a data warehouse that mirrors a single OLTP database).
- A central coordinator generates a global ordering (as in [72]).
- A distributed consensus protocol generates a global ordering (as in [47]).
- A deterministic scheme produces a global ordering. For example, each update producer generates timestamps locally and identical timestamps in a global view are settled with a deterministic tiebreaker like the producer's IP address.

Serial execution also needs a mechanism to provide consistent delivery of updates from multiple producers. Before completing a read, source nodes must not only ensure that all prior pending writes have been completed, but also that all notifications for prior updates have been received.

A simple solution is to channel all updates through a single server. This has the advantage of also providing a global ordering over all updates. However, a single-server solution creates a scalability bottleneck. Alternative solutions like broadcasting updates or periodic commits are possible, but introduce considerable synchronization overheads.

Speculative Execution with Deltas. Rather, DBToaster favors the use of speculative and optimistic processing techniques for it's runtime's execution protocol. The key insight here is that DBToaster's computations are all based on incremental processing. Unlike the standard usage of speculative execution, any work done speculatively need not be thrown away entirely – rather any work done can be revised through increments or deltas, to arrive at the final desired outcome.

In particular, a node can optimistically perform reads virtually immediately; it need not block on the vague possibility of a potential future write, although it could potentially be beneficial to block on pending write operations which the node is already aware of.

Avoiding blocking on potential future writes eliminates significant synchronization overheads, but out-of-order updates can cause the atomicity and desired ordering properties of trigger execution to be lost. DBToaster favor this point in the design space, since out-of-order events are expected to occur infrequently.

Furthermore, such events are likely to interfere with only a handful of prior updates: for example a write on one map entry followed by an out-of-order read on a different entry in the same map do not cause a problem. Finally, because write operations are limited to additive deltas, there is a clear mechanism for composing out-of-order writes.

Concretely, the effect of atomicity on a trigger program is restored by using a timestamping mechanism (one of the several options described above) to establish a *canonical* order of operations between the updates.

However, without synchronization measures, it is still possible for updates to arrive out of this newly defined canonical order. Frequently, this will not be an issue – a write on one map entry followed by an out-of-order read on a different entry in the same map do not cause a problem. Furthermore, because write operations are limited to additive deltas, there is a clear mechanism for composing out-of-order write operations.

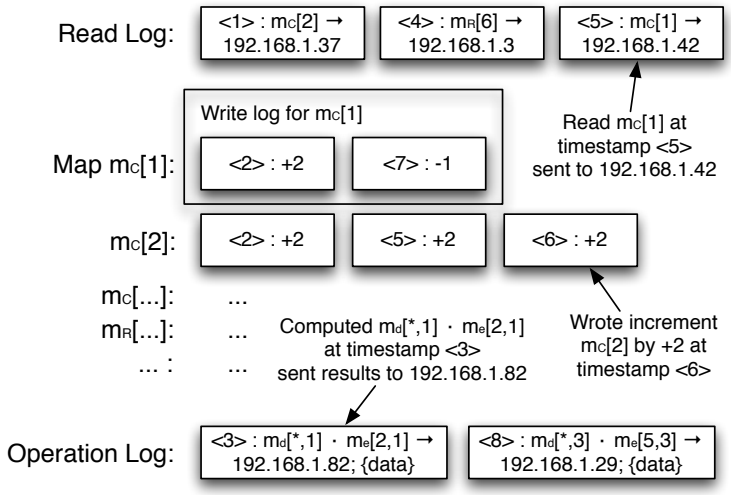


Figure 5.9: Supplemental data structures used to facilitate speculative execution in a distributed DDMS.

5.5.3 Out-of-Order Processing

Two types of out-of-order operations can occur in the speculative execution model: write-before-read, and read-before-write. We supplement maps with three additional data structures capturing timestamp information for operations, as illustrated in Figure 5.9:

- **Read Log**: Source nodes maintain a record of all read operations performed on their local maps. This information is supplemented with the identifier the compute node(s) which employ the data being read.
- **Operation Log**: Compute nodes maintain a record of all compute tasks which they perform. This information is supplemented with references to cached copies of all data received as part of this computation.
- **Write Log**: Destination nodes maintain a record of all write operations performed – in effect it can reconstruct the state of a map (to the best of its

knowledge) at any arbitrary timestamp. A compact representation with this property is a map that stores a log of (timestamp <t>, value) pairs, rather than individual values.

Out-of-order Reads. In the case of an out-of-order read operation (i.e., one that arrives after a write operation that logically precedes it), the write log makes it possible to reconstruct the state of the map at an earlier point in time.

Example 5.5.1 *Given the initial state in Figure 5.9, an update that requires a read on entry* m_C *[2] arrives with timestamp <3>.*

The value sent to the computation nodes is not the most recent value of the entry (m_C [2] = 6 *for all timestamps after <6>), but rather the sum of all values with lower timestamps (*m_C [2] = 2 *for timestamps <3>,<4>, and <5>).*

Out-of-order Writes. In the case of an out-of-order write operation, the read log allows us to send a *revising update* to each computation node affected by the write – which can then correct its computation by using the operation log.

Example 5.5.2 *Given the initial state in Figure 5.9, an update that requires a write on entry* m_R *[6] arrives with timestamp <3>.*

The value will be written as normal (i.e., inserted into the write log for m_R *[6], in sorted timestamp order). Additionally, because the read log shows a read on the same entry with a later timestamp, a corrective update will be sent to the computation node(s) to which the entries were originally sent to.*

Each affected computation node will consult its local operation log, modify the input data accordingly, repeat the computation, and forward the revising update to the

corresponding destination node(s).

Note that the operation log process could potentially be optimized even further by applying the DBToaster compilation algorithm to the computation itself; the revising update can essentially be thought of as a delta to the output of the computation.

Garbage Collection. All three data structures grow over time. To prevent unbounded memory usage, it is necessary to periodically truncate, or garbage collect the entries in each. This in turn, requires the runtime to periodically identify a cutoff point, the "last" update for which there are no operations pending within the cluster. The read history is truncated at this point, and all writes before this point are coalesced into a single entry. Though this process is slow, it does not interfere with any node's normal operations, and can be performed infrequently, for example once every few seconds.

Hybrid Consistency. While the speculative execution model and its eventually consistent results are advantageous from a performance and scalability perspective, there may not be a point at which the state of all maps in the system corresponds to a consistent snapshot of a transition programs evaluated over any prefix of the update stream.

That is, there is no guarantee that the system has actually converged to its eventually consistent state in the presence of a highly dynamic update stream.

However, a side effect of the garbage collection process is that each garbage collection run, in effect generates a consistent snapshot of the system. As in other eventual consistency systems [84], this approach offers a hybrid consistency model, specifically the same infrastructure produces both low-latency

eventually consistent results, as well as higher-latency consistent snapshots.

5.5.4 Partitioning Schemes

The second challenge associated with distributing a transition program across the cluster is the distribution of logical nodes (source, computation, and destination) across physical hardware in the cluster.

In addition to more complex, min-cut based partitioning schemes for the data, DBToaster considers two simple partitioning heuristics for distributing computation:

1. data shipping: evaluate program statements where their results will be stored, at destination nodes, or
2. program shipping: evaluate program statements where their input maps are stored, at source nodes.

Data-Shipping. Given the one-to-one correspondence between computation nodes and destination nodes, the simplest partitioning scheme is to perform computations where the data will be stored – that is, to co-locate the destination and computation nodes. As part of update evaluation, each source node transmits all relevant map entries to the destination node. Upon arrival, the destination node evaluates the statement and stores the result.

Program-Shipping. Though simple, transmitting every relevant map entry with every update can be wasteful, especially if map entries on the source side don't change frequently. An alternative approach is to co-locate all of the source nodes

and the computation node. When evaluating an update, the computation can be performed instantaneously, and the only overhead is transmitting the result(s) to the destination node(s). Program-shipping is particularly effective in queries where update effects are small (e.g., queries consisting mostly of equijoins on key columns).

However, program-shipping approach introduces an additional complication. It is typically not possible to generate a partitioning of the data that ensures that for each statement in a trigger program, all the source nodes will be co-located. In order to achieve a partitioning, data must be replicated; each map is stored on multiple physical nodes. While replication is typically a desirable characteristic, storage-constrained infrastructures may need to use a more complex partitioning scheme.

5.6 Evaluation

The DBToaster implementation consists of a compiler and a runtime: The compiler produces C++ or OCaml code suitable for standalone use with the runtime component, or for embedding into another application.

A thorough evaluation of the DBToaster concept was originally performed by Ahmad and Koch and has not been published. Substantial improvements and generalizations to DBToaster's formalisms and compilation process have been achieved since then. However, the original experiments remain valid for the outputs produced by DBToaster. These initial results are presented here.

The evaluation was performed on a Dual Intel Xeon 5335 machine with 8

cores running at 2.0 GHz, and 16 GB of RAM, running Ret Hat Enterprise Linux 4 (Kernel 2.6.18).

DBToaster's performance was tested on the single threaded C++ code it generates, compiled with g++ 4.3.2 with optimization level "-O3" and flags for aggressive inlining and loop optimizations. The evaluation compared DBToaster against equivalent queries run on Postgres 8.3.7, as well as code produced by a non-incremental version of DBToaster that processes queries all at once, rather than by using an incrementally maintained agile view.

VWAP. The first comparison point was a parameterized version of the VWAP query presented in Example 1.2.1. The VWAP query was run over historical data for the MSFT (Microsoft) ticker symbol from the TotalView-ITCH [1] historical order book, representing the NASDAQ stock exchange over the three month period from December 2008 to February 2009.

DBToaster's incremental strategy ensures that query results are updated with each update. For the non-incremental evaluation strategies that DBToaster was compared against, the number of times over the simulated three month period that the query results are refreshed was varied. Figure 5.10 illustrates the trade-off point for VWAP: The cost of fully re-evaluating the query about 25 times over the three month period begins to exceed the overheads of incremental maintenance. Figure 5.11 shows this same data and provides a comparison with Postgres.

Star Schema Benchmark. The second evaluation point is the Star Schema Benchmark [71] (SSB), a denormalized form of the TPC-H [86] benchmark, intended to be representative of data warehousing workloads. The experiment is based on

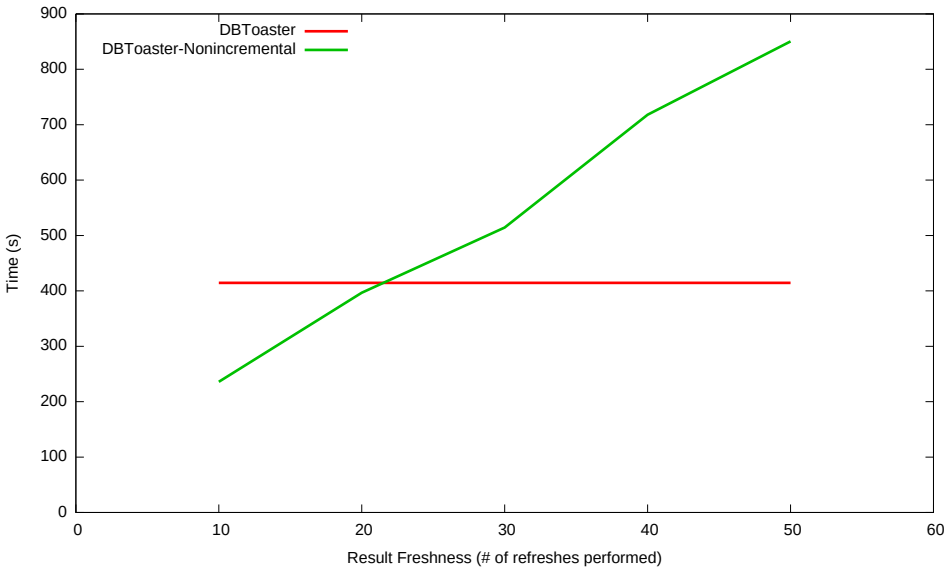


Figure 5.10: Tradeoff between Batch Processing and Incremental Computation on VWAP

Number of Refreshes	DBToaster	DBToaster-Nonincremental	Postgres
10	414.5	236.0	1712.7
20	414.5	396.8	3156.4
30	414.5	514.4	4580.2
40	414.5	718.0	6070.8
50	414.5	850.3	7520.3

Figure 5.11: Time to complete the VWAP query relative to the number of requests posed over the dataset

loading a data warehouse – input data taken from the TPC-H benchmark data generator is first prepared with a data cleaning and transformation query and immediately analyzed with SSB query 4.1, a join-aggregate query that computes per-year profit for several nations.

As before, the refresh rate is varied for the non-incremental systems. The tradeoff point is presented in Figure 5.12 and the full set of results are presented in Figure 5.13.

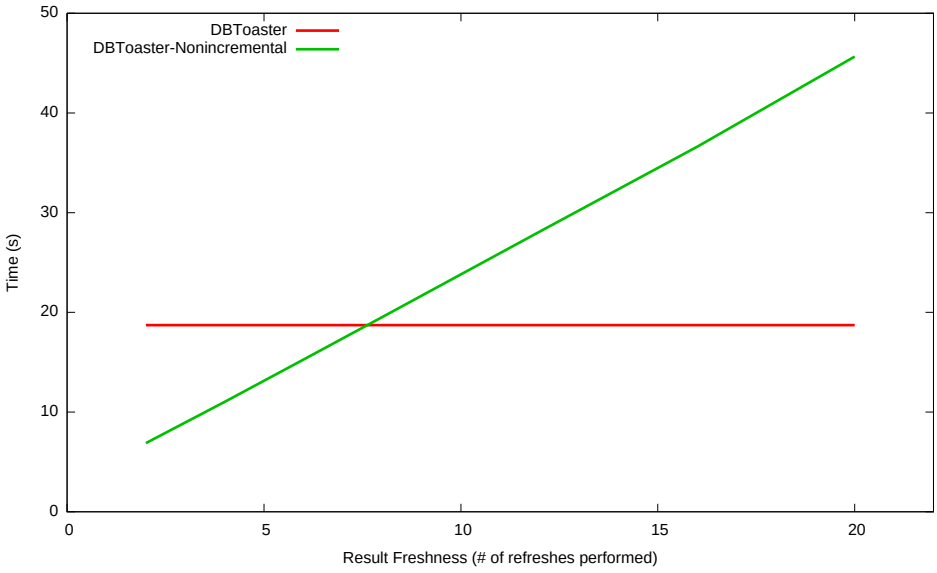


Figure 5.12: Tradeoff between Batch Processing and Incremental Computation on the star schema benchmark

Number of Refreshes	DBToaster	DBToaster-Nonincremental	Postgres
2	18.72	6.89	348.40
4	18.72	11.02	700.97
6	18.72	15.26	1053.63
8	18.72	19.53	1411.34
12	18.72	28.09	2131.52
16	18.72	36.61	2856.84
20	18.72	45.65	3612.32

Figure 5.13: Time to complete the star schema query relative to the number of requests posed over the dataset

CHAPTER 6

CONCLUSIONS

In the words of Jim Gray, "The world of science has changed, and there is no question about this. The new model is for the data to be captured by instruments or generated by simulations before being processed by software and for the resulting information or knowledge to be stored in computers. Scientists only get to look at their data fairly late in the pipeline. The techniques and technologies for such data-intensive science are so different that it is worth distinguishing data-intensive science from computational science as a new, *fourth paradigm* for scientific exploration" [40]

As computer scientists, it is our responsibility to step up to this challenge of providing scientists and researchers with *usable* and *relevant* tools for managing, understanding, and analyzing large amounts of data. In my research, I have addressed two key challenges in this grand undertaking: (1) Providing tools for dealing with uncertainty in data, and (2) Providing tools for complex monitoring of large, frequently changing state.

Grey-Box Probabilistic Databases. Probabilistic databases are intended generalize the analysis of probabilistic data. This sort of generalization enables the application of traditional database techniques (as well as technologies) to a much broader range of scientific problems, which in turn improves the efficiency and feasibility of large scale scientific data analysis of probabilistic data.

The use of stochastic black-box functions to define probability distributions has been a major step forward for the application of probabilistic databases to real-world problems. As they allow the integration of virtually any distribution

into a powerful data-processing framework, these VG-Functions have shown themselves to be extremely relevant, but not sufficiently efficient to be useful in a broad range of contexts.

By carefully providing users with the *option* of removing bits of the opacity associated with these black-box functions, we can turn them into grey-box functions. The additional understanding about the function's characteristics (whether derived from user input, or programmatically from the function itself) provides users with a clean and relevant interface for building models over arbitrary probability distributions, while still retaining enough efficiency to be useable.

PIP. PIP was the first instantiation of a grey-box probabilistic database. PIP uses a symbolic representation for data, which completely characterizes the effects of a query on the uncertainty in the query's inputs. In this way, it is able to exploit useful characteristics of the distributions being queried in order to improve performance and accuracy for a wide range of queries – even with a very limited and simplistic set of statistical tools.

Symbolic representations of uncertainty like C-Tables can be used to make the computation of expectations and other statistical measures in probabilistic databases more accurate and more efficient. The availability of the expression being measured enables a broad range of sampling techniques that rely on this information and allows more effective selection of the appropriate technique for a given expression.

Jigsaw. is a powerful tool for evaluating and optimizing parameterized what-if scenarios. Jigsaw efficiently performs parameter optimization, allows online ex-

ploration of a scenario's parameter space at interactive speeds, and rapid evaluation of a common class of Markovian processes.

The key to these three processes is a novel "fingerprinting" mechanism which identifies correlations between similar, yet distinct probability distributions. Fingerprints can be applied to several common tasks that arise in the domain of cloud service management, and demonstrated that Jigsaw can achieve speedups of as much as 2 orders of magnitude.

DDMS. I have described a new class of data management system, designed for monitoring large, frequently changing state: the Dynamic Data Management System (DDMS). Related to both stream processors and incremental view maintenance, a DDMS is neither. Unlike stream processors, a DDMS supports complex queries with persistent state. Unlike incremental view maintenance in database systems, a DDMS supports high update rates.

The DBToaster compiler is part of a greater effort towards making DDMS a reality. By using compilation techniques based on a recursive re-interpretation of incremental view-maintenance, DBToaster is able to efficiently process ordinary SQL queries incrementally. With not only the volume, but also the rate of data used in modern scientific applications growing, as data storage becomes progressively, DBToaster fills a vital niche in promoting research.

Both uncertainty and high-data-rate computations are critical challenges in the world of science. I believe that these three projects are concrete steps towards realizing Jim Gray's vision of using computer science to support data-intensive research, and will form the foundation for a wide array of scientific tools.

APPENDIX A

PIP QUERY Q3

```
create table 'shipping_params' as
  select
    avg   (l_shipdate    - o_orderdate) as ship_mu,
    avg   (l_receiptdate - l_shipdate ) as arrv_mu,
    stddev(l_shipdate    - o_orderdate) as ship_sigma,
    stddev(l_receiptdate - l_shipdate ) as arrv_sigma,
    l_partkey as p_partkey
  from orders,lineitem
  where o_orderkey = l_orderkey
  group by partkey;
alter table params add constraint "p_partkey_pkey"
  primary key (p_partkey);
-- BEGIN QUERY --
create temporary table q3_shipping as
  select o_orderkey AS orderkey, o_custkey AS custkey,
    CREATE_VARIABLE('Normal',row(ship_mu,ship_sigma))
    CREATE_VARIABLE('Normal',row(arrv_mu,arrv_sigma))
  from  orders,lineitem,shipping_params
  where p_partkey = l_partkey;
   and  o_orderdate = today()
   and  o_orderkey  = l_orderkey;
create temporary table q3_annoyed as
  select custkey from q3_shipping where ship > 120
union all
```

```
  select custkey from q3_shipping where arrv > 90;
create temporary table q3_order_increase as
  select o_orderkey, o_custkey,
     CREATE_VARIABLE('Poisson', row(increase)) *
       l_extended_price * (1.0 - l_discount) as rev
  from (select newc / oldc as increase, custkey
    from (select o_custkey as custkey,
              sum(o_orderdate.year-1996.0) AS newc,
              sum(1997.0-o_orderdate.year) AS oldc
      where o_orderdate.year = 1997
        or  o_orderdate.year = 1996
      group by custkey
     ) as counts
   ) as increase_per_cust,
    orders
  where custkey = o_custkey
 ) as var_increase_per_customer,
 (select lineitem.*,
  from  nation,supplier, lineitem, partsupp
  where n_name = 'japan' and n_nationkey = s_nationkey
   and  s_suppkey = ps_suppkey
   and  ps_partkey = l_partkey
   and  ps_suppkey = l_suppkey
 ) as items_from_japan;
-- BEGIN SAMPLING --
select avg(confidence),
```

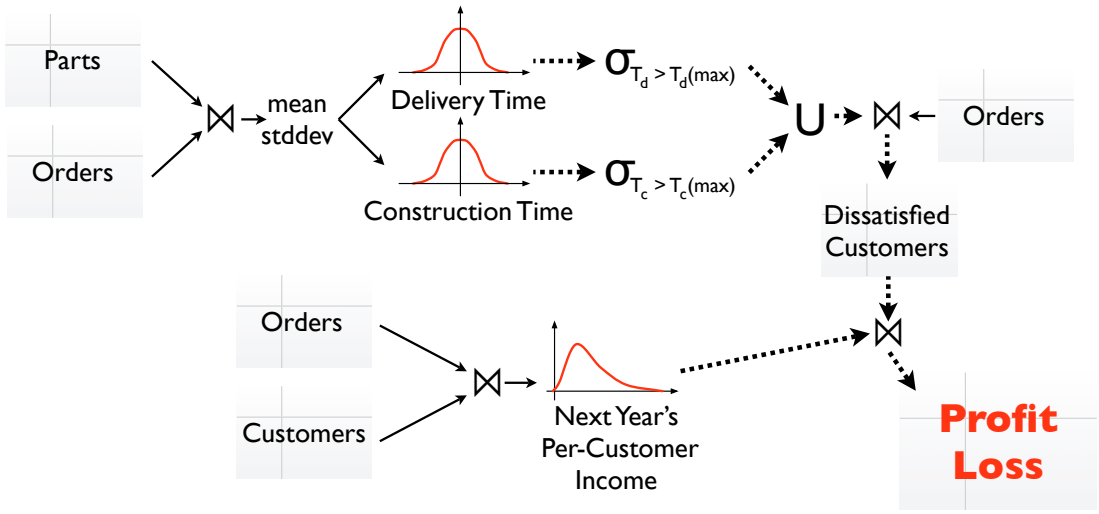


Figure A.1: A dataflow diagram of query 3. Dotted lines represent probabilistic data.

```
        expected_sum(rev, q3_annoyed)
from    q3_annoyed,
        (select o_custkey as custkey, rev
         from   q3_revenue_gains
        ) as revenues
where   revenues.custkey = q3_annoyed.custkey;
```

Query 3 is based on [46]'s Q1 and Q2 and represented graphically in Figure A.1. A prebuilt table of shipping and construction times is used to parametrize Normal distributions that predict time from order to shipment and time from shipment to arrival.

Note the use of the CREATE_VARIABLE function to create these variables. This table is compared against arbitrary customer satisfaction thresholds to generate a probabilistic table containing a customer if the customer was dissatisfied

with the shipping times (we selected thresholds that left an average of 10% of customers dissatisfied by at least one event).

Also note that probabilistic variables may be used in WHERE clauses in the same way as deterministic ones.

Separately, the query computes the expected profit for each customer and joins it with the table of dissatisfied customers to estimate the amount of profit lost in the coming year.

APPENDIX B

A DBTOASTER WORKFLOW EXAMPLE

Recall the simple query defined in **ToaStQL** in Example 5.3.1:

```
CREATE TABLE R(A int, B int)
  FROM FILE 'test/data/r.dat' LINE DELIMITED
  CSV (delimiter := ',', schema := 'int,int',
       eventtype := 'insert');
CREATE TABLE S(B int, C int)
  FROM FILE 'test/data/s.dat' LINE DELIMITED
  CSV (delimiter := ',', schema := 'int,int',
       eventtype := 'insert');
CREATE TABLE T(C int, D int)
  FROM FILE 'test/data/t.dat' LINE DELIMITED
  CSV (delimiter := ',', schema := 'int,int',
       eventtype := 'insert');
SELECT sum(A*D) FROM R,S,T WHERE R.B=S.B AND S.C=T.C;
```

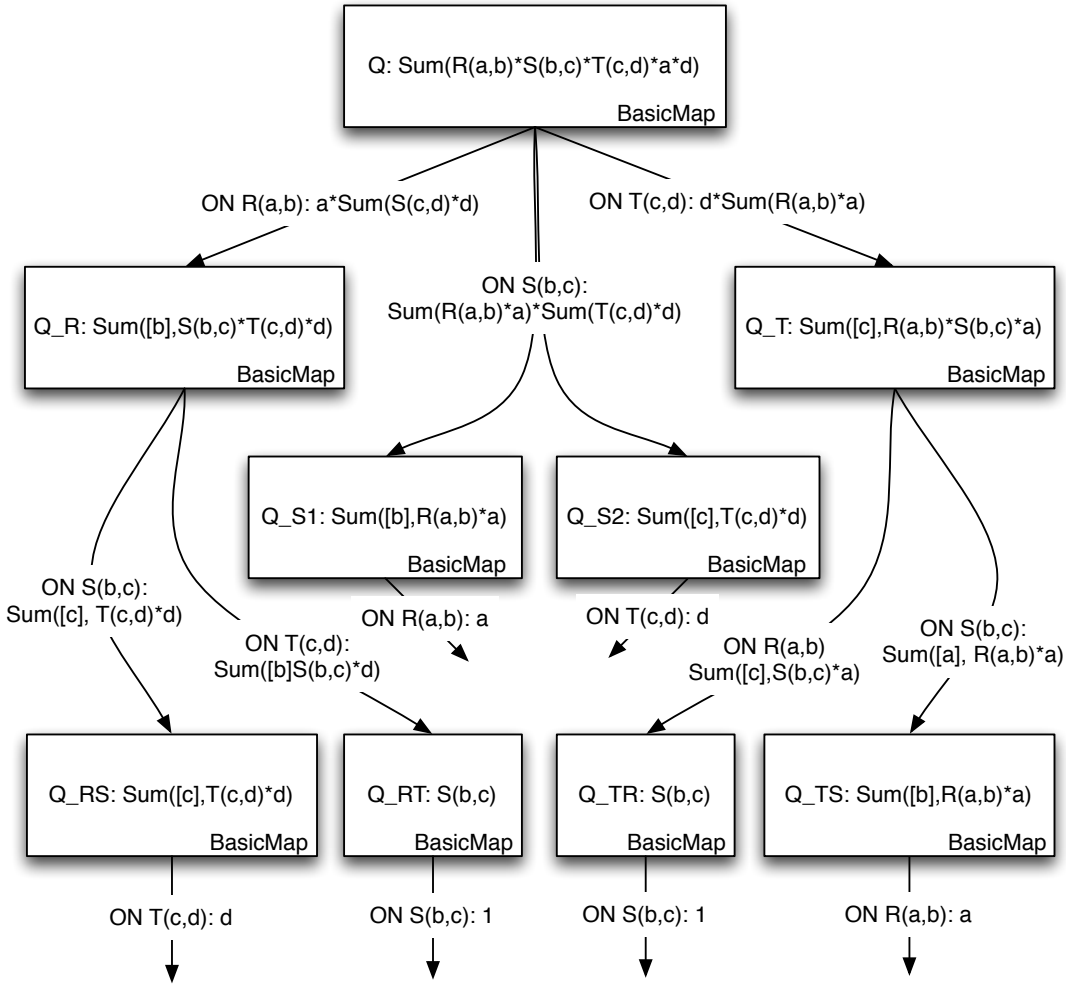
As in the example, the `CREATE TABLE` statements are set aside for later use. Meanwhile, the `SELECT` query is rewritten into a **DBToaster relational calculus** `AggSum` by multiplying the query's `FROM`, target, and `WHERE` clauses.

```
AggSum([], R[R_A,R_B] * S[S_B,S_C] * T[T_C,T_D] * R_A * T_D *
           (R_B = S_B) * (S_C = T_C))
```

After a simplification step, this becomes (with variable names substituted for readability):

```
AggSum([], R[A,B] * S[B,C] * T[C,D] * A * D)
```

Following the process illustrated in Example 5.2.1 and discussed further in Section 5.3.3, this expression is compiled into an **incremental plan**, as illustrated in this diagram:



The materializing optimizer runs on this plan. It will notice several identical maps, namely that $Q_{RT} \equiv Q_{TR}$, $Q_{S1} \equiv Q_{TS}$, and $Q_{S2} \equiv Q_{RS}$. For each equivalent pair, one will be replaced by the other. For this query, no further optimizations are performed.

The optimized incremental plan will be reduced down to a **program sketch** as follows:

```
TABLES { R: [...], S: [...], T: [...] }
MAPS {
  Q[][],
  Q_R[][B]
  Q_T[][C]
  Q_RS[][C]
  Q_RT[][B,C]
  Q_TS[][B]
}
EVENTS {
  +R(A,B)
    Q[][]       += A * Q_R[][B]
    Q_T[][C]    += A * Q_RT[][B,C]
    Q_TS[][B]   += A
  +S(B,C)
    Q[][]       += Q_TS[][B] * Q_RS[][C]
    Q_R[][B]    += Q_RS[][C]
    Q_T[][C]    += Q_TS[][B]
    Q_RT[][B,C] += 1
  +T(C,D)
    Q[][]       += D * Q_T[][C]
    Q_R[][B]    += D * Q_RT[][B,C]
    Q_SR[][C]   += D
}
```

For brevity, only the insertion events are shown. Deletion events are implemented as the negation of the above expressions, while update events are implemented as a deletion followed by an insertion. This program sketch is translated into K3, an excerpt of which (for insertions of R) is shown below:

```
ON_insert_R(QUERY_1_1S1R_A,QUERY_1_1S1R_R__B)
PCValueUpdate(SingletonPC("QUERY_1_1"),[],[],
  Apply(
    Lambda(AVar("current_v",TFloat),
      Add(Var("current_v"),
        Mult(Var("QUERY_1_1S1R_A"),
          Apply(
            Lambda(AVar("slice",Collection(TTuple(TFloat ; TFloat))),
              IfThenElse(Member(Var("slice"),[Var("QUERY_1_1S1R_R__B");]),
                Lookup(Var("slice"),[Var("QUERY_1_1S1R_R__B");]),
                Apply(
                  Lambda(AVar("init_val",TFloat),
                    Block(
                      [PCValueUpdate(OutPC("QUERY_1_1R1"),[],
                         [Var("QUERY_1_1S1R_R__B");],Var("init_val"));
                       Var("init_val");])),Const(CFloat(0.)))))),
            OutPC("QUERY_1_1R1"))))),SingletonPC("QUERY_1_1")))
Iterate(
  Lambda(ATuple(["QUERY_1_1T_T__C",TFloat;"updated_v",TFloat]),
    PCValueUpdate(OutPC("QUERY_1_1T1"),[],[Var("QUERY_1_1T_T__C");],
      Var("updated_v"))),
  Apply(
    Lambda(AVar("current_slice",Collection(TTuple(TFloat ; TFloat))),
      Map(
        Lambda(ATuple(["QUERY_1_1T_T__C",TFloat;"dv",TFloat]),
          IfThenElse(Member(Var("current_slice"),[Var("QUERY_1_1T_T__C");]),
            Tuple(Var("QUERY_1_1T_T__C"),
              Add(Lookup(Var("current_slice"),[Var("QUERY_1_1T_T__C");]),
```

```
            Var("dv"))),
          Tuple(Var("QUERY_1_1T_T__C"),Add(Const(CFloat(0.)),Var("dv")))))),
      GroupByAggregate(
        AssocLambda(ATuple(["QUERY_1_1S1R_R__B",TFloat;
                            "QUERY_1_1T_T__C",TFloat;"v",TFloat]),
                    AVar("accv",TFloat),
          Add(Var("v"),Var("accv"))),Const(CFloat(0.)),
        Lambda(ATuple(["QUERY_1_1S1R_R__B",TFloat;"QUERY_1_1T_T__C",TFloat;
                       "v",TFloat]),
          Tuple(Var("QUERY_1_1T_T__C"))),
        Map(
          Lambda(ATuple(["QUERY_1_1S1R_R__B",TFloat;"QUERY_1_1T_T__C",TFloat;
                         "v2",TFloat]),
            Tuple(Var("QUERY_1_1S1R_R__B"),Var("QUERY_1_1T_T__C"),
              Mult(Var("QUERY_1_1S1R_A"),Var("v2")))),
          Apply(
            Lambda(AVar("slice",Collection(TTuple(TFloat ; TFloat ; TFloat))),
              Slice(Var("slice"),[Var("QUERY_1_1S1R_R__B");])),
            OutPC("QUERY_1_1R1T1")))))),OutPC("QUERY_1_1T1")))
PCValueUpdate(OutPC("QUERY_1_1S1"),[],[Var("QUERY_1_1S1R_R__B");],
  IfThenElse(Member(OutPC("QUERY_1_1S1"),[Var("QUERY_1_1S1R_R__B");]),
    Apply(
      Lambda(AVar("current_v",TFloat),
        Add(Var("current_v"),Var("QUERY_1_1S1R_A"))),
      Lookup(OutPC("QUERY_1_1S1"),[Var("QUERY_1_1S1R_R__B");])),
    Add(Var("QUERY_1_1S1R_A"),Const(CFloat(0.)))))
```

Next, the K3 optimizer iterates over the code.

```
ON_insert_R(QUERY_1_1S1R_A,QUERY_1_1S1R_R__B)
IfThenElse(Member(OutPC("QUERY_1_1R1"),[Var("QUERY_1_1S1R_R__B");]),
  PCValueUpdate(SingletonPC("QUERY_1_1"),[],[],
    Apply(
```

```
      Lambda(AVar("current_v",TFloat),
        Add(Var("current_v"),
          Mult(Var("QUERY_1_1S1R_A"),
            Lookup(OutPC("QUERY_1_1R1"),[Var("QUERY_1_1S1R_R__B");]))))),
      SingletonPC("QUERY_1_1"))),
  PCValueUpdate(SingletonPC("QUERY_1_1"),[],[],
    Apply(
      Lambda(AVar("current_v",TFloat),
        Add(Var("current_v"),
          Mult(Var("QUERY_1_1S1R_A"),
            Apply(
              Lambda(AVar("init_val",TFloat),
                Block(
                  [PCValueUpdate(OutPC("QUERY_1_1R1"),[],
                     [Var("QUERY_1_1S1R_R__B");],Var("init_val"));Var("init_val");])),
              Const(CFloat(0.)))))),SingletonPC("QUERY_1_1"))))
Iterate(
  Lambda(ATuple(["QUERY_1_1T_T__C",TFloat;"updated_v",TFloat]),
    PCValueUpdate(OutPC("QUERY_1_1T1"),[],[Var("QUERY_1_1T_T__C");],
      Var("updated_v"))),
  Map(
    Lambda(ATuple(["QUERY_1_1T_T__C",TFloat;"dv",TFloat]),
      IfThenElse(Member(OutPC("QUERY_1_1T1"),[Var("QUERY_1_1T_T__C");]),
        Tuple(Var("QUERY_1_1T_T__C"),
          Add(Lookup(OutPC("QUERY_1_1T1"),[Var("QUERY_1_1T_T__C");]),Var("dv"))),
        Tuple(Var("QUERY_1_1T_T__C"),Add(Const(CFloat(0.)),Var("dv"))))),
    GroupByAggregate(
      AssocLambda(ATuple(["QUERY_1_1S1R_R__B",TFloat;"QUERY_1_1T_T__C",TFloat;
                          "v2",TFloat]),AVar("accv",TFloat),
        Add(Mult(Var("QUERY_1_1S1R_A"),Var("v2")),Var("accv"))),
      Const(CFloat(0.)),
      Lambda(ATuple(["QUERY_1_1S1R_R__B",TFloat;"QUERY_1_1T_T__C",TFloat;"v2",TFloat]),
        Tuple(Var("QUERY_1_1T_T__C"))),
```

```
      Slice(OutPC("QUERY_1_1R1T1"),[Var("QUERY_1_1S1R_R__B");])))))
IfThenElse(Member(OutPC("QUERY_1_1S1"),[Var("QUERY_1_1S1R_R__B");]),
  PCValueUpdate(OutPC("QUERY_1_1S1"),[],[Var("QUERY_1_1S1R_R__B");],
    Apply(
      Lambda(AVar("current_v",TFloat),
        Add(Var("current_v"),Var("QUERY_1_1S1R_A"))),
      Lookup(OutPC("QUERY_1_1S1"),[Var("QUERY_1_1S1R_R__B");]))),
  PCValueUpdate(OutPC("QUERY_1_1S1"),[],[Var("QUERY_1_1S1R_R__B");],
    Add(Var("QUERY_1_1S1R_A"),Const(CFloat(0.)))))
```

Finally, the K3 program is compiled directly into OCaml or C++.

BIBLIOGRAPHY

[1] NASDAQ TotalView order book, http://www.nasdaqtrader.com/Trader.aspx?id=TotalVi

[2] Postgresql. http://www.postgresql.org/.

[3] The R-Project for statistical computing. http://www.r-project.org.

[4] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal: The International Journal on Very Large Data Bases*, 2003.

[5] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proc. of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 147–160, 2008.

[6] Yanif Ahmad, Olga Papaemmanouil, Ugur Çetintemel, and Jennie Rogers. Simultaneous equation systems for query processing on continuous-time data streams. In *ICDE*, pages 666–675. IEEE, 2008.

[7] N. Ahmed, T. Natarajan, and K.R. Rao. Discrete cosine transfom. *Computers, IEEE Transactions on*, C-23(1):90 – 93, jan. 1974.

[8] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. "Fast and Simple Relational Processing of Uncertain Data". In *Proc. ICDE*, 2008.

[9] Lyublena Antova, Christoph Koch, and Dan Olteanu. "10^{10^6} Worlds and Beyond: Efficient Representation and Processing of Incomplete Information". In *Proc. ICDE*, 2007.

[10] Lyublena Antova, Christoph Koch, and Dan Olteanu. MayBMS: Managing incomplete information with probabilistic world-set decompositions. In *ICDE*, 2007.

[11] David A. Basin, Felix Klaedtke, and Samuel Müller. Policy monitoring in first-order temporal logic. In *CAV*, pages 1–18, 2010.

[12] Holger Bast and Ingmar Weber. The completesearch engine: Interactive, efficient, and towards ir& db integration. In *CIDR*, pages 88–95, 2007.

[13] Randall G. Bello, Karl Dias, Alan Downing, James J. Feenan Jr., James L. Finnerty, William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized views in oracle. In *Proc. of the 24th International Conference on Very Large Data Bases (VLDB'98)*, pages 659–664. Morgan Kaufmann, 1998.

[14] A. Bialecki, M. Cafarella, D. Cutting, and O. OMalley. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene. apache. org/hadoop*, 2005.

[15] G.E.P. Box and Mervin E. Muller. A note on the generation of random normal deviates. *Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[16] Lars Brenna, Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker M. White. Cayuga: a high-performance event processing engine. In *Proc. of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 1100–1102, 2007.

[17] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.

[18] Stefano Ceri, Roberta Cochrane, and Jennifer Widom. Practical applications of triggers and constraints: Success and lingering issues (10-year award). In *VLDB*, pages 254–262, 2000.

[19] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[20] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[21] Reynold Cheng, Dmitri V. Kalashnikov, and Sunil Prabhakar. "Evaluating Probabilistic Queries over Imprecise Data". In *Proc. SIGMOD*, pages 551–562, 2003.

[22] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 469–480, 1996.

[23] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[24] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO*, pages 266–296, 2006.

[25] Nilesh Dalvi and Dan Suciu. "Efficient query evaluation on probabilistic databases". *VLDB Journal*, **16**(4):523–544, 2007.

[26] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[27] Amol Deshpande and Samuel Madden. MauveDB: supporting model-based user views in database systems. In *ACM SIGMOD*, 2006.

[28] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986.

[29] Yupeng Fu, Keith Kowalczykowski, Kian Win Ong, Yannis Papakonstantinou, and Kevin Keliang Zhao. Ajax-based report pages as incrementally rendered views. In *SIGMOD Conference*, pages 567–578, 2010.

[30] Thanaa M. Ghanem, Ahmed K. Elmagarmid, Per-ke Larson, and Walid G. Aref. Supporting views in data stream management systems. *ACM Trans. Database Syst.*, 35(1), 2010.

[31] Michael Gibas, Ning Zheng, and Hakan Ferhatosmanoglu. A general framework for modeling and processing optimization queries. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1069–1080. VLDB Endowment, 2007.

[32] W.R. Gilks, S. Richardson, and David Spiegelhalter. *Markov Chain Monte Carlo in Practice: Interdisciplinary Statistics*. Chapman and Hall/CRC, 1995.

[33] Todd J. Green and Val Tannen. "Models for Incomplete and Probabilistic

Information". In *International Workshop on Incompleteness and Inconsistency in Databases (IIDB)*, 2006.

[34] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *Proc. of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 328–339, 1995.

[35] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: database-supported program execution. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD Conference*, pages 1063–1066. ACM, 2009.

[36] Alfred Haar. Zur theorie der orthogonalen funktionensysteme. *Mathematische Annalen*, 69:331–371, 1910. 10.1007/BF01456326.

[37] Peter J. Haas, Christopher M. Jermaine, Subi Arumugam, Fei Xu, Luis Leopoldo Perez, and Ravi Jampani. MCDB-R: Risk analysis in the database. *PVLDB*, 3(1):782–793, 2010.

[38] D. Hamlet. *Encyclopedia of Software Engineering*, chapter Random testing, pages 970–978. Wiley, New York, 1994.

[39] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *SIGMOD Conference*, pages 171–182, 1997.

[40] A.J.G. Hey, S. Tansley, and K.M. Tolle. *The fourth paradigm: data-intensive scientific discovery*. Microsoft Research, 2009.

[41] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. Range queries in olap data cubes. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 73–88, New York, NY, USA, 1997. ACM.

[42] Rob Iati. The Real Story of Trading Software Espionage, AdvancedTrading.com,july 10, 2009.

[43] T. Imielinski and W. Lipski. "Incomplete information in relational databases". *Journal of ACM*, **31**(4):761–791, 1984.

[44] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building

blocks. In Paulo Ferreira, Thomas R. Gross, and Luís Veiga, editors, *EuroSys*, pages 59–72. ACM, 2007.

[45] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stanley B. Zdonik. Towards a streaming sql standard. *PVLDB*, 1(2):1379–1390, 2008.

[46] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher M. Jermaine, and Peter J. Haas. MCDB: a monte carlo approach to managing uncertain data. In *ACM SIGMOD*, 2008.

[47] Flavio P. Junqueira and Benjamin C. Reed. The life and times of a zookeeper. In *PODC*, New York, NY, USA, 2009.

[48] Gershon Kedem. Automatic differentiation of computer programs. *ACM Trans. Math. Softw.*, 6:150–165, June 1980.

[49] Oliver Kennedy, Yanif Ahmad, and Christoph Koch. Dbtoaster: Agile views for a dynamic data management system. CIDR, 2011.

[50] Oliver Kennedy and Christoph Koch. PIP: A database system for great and small expectations. In *ICDE*, 2010.

[51] Oliver Kennedy, Steve Lee, Charles Loboz, Slawek Smyl, and Suman Nath. Fuzzy prophet: Parameter exploration in uncertain enterprise scenarios. In *ACM SIGMOD*, 2011.

[52] Oliver Kennedy and Suman Nath. Jigsaw: Efficient optimization over uncertain enterprise data. In *ACM SIGMOD*, 2011.

[53] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice–Hall, 1988.

[54] Christoph Koch. "MayBMS: A system for managing large uncertain and probabilistic databases". In Charu Aggarwal, editor, *Managing and Mining Uncertain Data*, chapter 6. Springer-Verlag, February 2009.

[55] Christoph Koch. Incremental query evaluation in a ring of databases. In *PODS*, pages 87–98, 2010.

[56] Harold J. Kushner and Dean S. Clark. *Stochastic approximation methods for constrained and unconstrained systems*. Springer-Verlag, 1978.

[57] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[58] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[59] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. McGraw-Hill Book Company, 1982.

[60] Jacob Lehman. Zombies vs shakespeare. First Performed by Ring of Steel: Ithaca, 2010.

[61] K. Lellahi and V. Tannen. A calculus for collections and aggregates. In *Category Theory and Computer Science*, pages 261–280. Springer, 1997.

[62] Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras, editors. *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. IEEE, 2010.

[63] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI*, 1988.

[64] M. Lutz. *Programming python*. O'Reilly Media, Inc., 2011.

[65] Erik Meijer, Brian Beckman, and Gavin M. Bierman. Linq: reconciling object, relations and xml in the .net framework. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *SIGMOD Conference*, page 706. ACM, 2006.

[66] N. Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44(335), 1949.

[67] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H.Teller, and Edward Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6), June 1953.

[68] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[69] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *Proc. of the 1999 USENIX Annual Technical Conference*, pages 183–192, 1999.

[70] Christopher Olston, Edward Bortnikov, Khaled Elmeleegy, Flavio Junqueira, and Benjamin Reed. Interactive analysis of web-scale data. In *CIDR*, 2009.

[71] Patrick O'Neil, Elizabeth O'Neil, and Xuedong Chen. The star schema benchmark. http://www.cs.umb.edu/ poneil/StarSchemaB.PDF , 2007.

[72] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. 2010.

[73] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer, Berlin, 1981.

[74] Christopher Ré and Dan Suciu. Materialized views in probabilistic databases: for information exchange and query optimization. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 51–62. VLDB Endowment, 2007.

[75] Christina Rockwell. Bolder, older, and selective: Factors of individual-specific foraging strategies in steller's jays. Master's thesis, Humboldt State University, 2010.

[76] Nick Roussopoulos. An incremental access method for viewcache: Concept, algorithms, and cost analysis. *ACM Transactions on Database Systems*, 16(3):535–563, 1991.

[77] Florin Rusu, Fei Xu, Luis Leopoldo Perez, Mingxi Wu, Ravi Jampani, Chris Jermaine, and Alin Dobra. The dbo database system. In *SIGMOD Conference*, pages 1223–1226, 2008.

[78] M. H. Safizadeh and B. M. Thornton. Optimization in simulation experiments using response surface methodology. *COMP. INDUST. ENG.*, 8(1):11–28, 1984.

[79] Prithviraj Sen and Amol Deshpande. "Representing and Querying Correlated Tuples in Probabilistic Databases". In *Proc. ICDE*, pages 596–605, 2007.

[80] Sarvjeet Singh, Chris Mayfield, Sagar Mittal, Sunil Prabhakar, Susanne E. Hambrusch, and Rahul Shah. Orion 2.0: native support for uncertain data. In *ACM SIGMOD*, 2008.

[81] National Snow and Ice Data Center/World Data Center for Glaciology. International ice patrol (iip) iceberg sightings database. Digital Media, 2008.

[82] Michael Stonebraker. The case for partial indexes. *SIGMOD Record*, 18(4):4–11, 1989.

[83] Gerald J. Sussman and Guy L. Steele Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.

[84] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182, New York, NY, USA, 1995. ACM.

[85] Arvind Thiagarajan and Samuel Madden. Querying continuous functions in a database system. In *ACM SIGMOD*, 2008.

[86] Transaction Processing Performance Council. *TPC Benchmark H (Decision Support)*, revision 2.8.0 edition, 2008. http://www.tpc.org/tpch/spec/tpch2.6.0.pdf.

[87] Andreas von Bechtoldsheim. Invited talk, EPFL, sept. 21, 2010.

[88] Daisy Zhe Wang, Eirinaios Michelakis, Minos Garofalakis, and Joseph M. Hellerstein. "BayesStore: Managing large, uncertain data repositories with probabilistic graphical models". In *VLDB '08, Proc. 34th Int. Conference on Very Large Databases*, 2008.

[89] Walker M. White, Mirek Riedewald, Johannes Gehrke, and Alan J. Demers. What is "next" in event processing? In *Proc. of the 26th ACM Symposium on Principles of Database Systems (PODS'07)*, pages 263–272, 2007.

[90] Michael L. Wick, Andrew McCallum, and Gerome Miklau. Scalable probabilistic databases with factor graphs and mcmc. *PVLDB*, 3(1), 2010.

[91] Jennifer Widom. "Trio: a system for data, uncertainty, and lineage". In Charu Aggarwal, editor, *Managing and Mining Uncertain Data*. Springer-Verlag, February 2009.

[92] Yi Zhang, Weiping Zhang, and Jun Yang. I/O-efficient statistical computing with RIOT. In Li et al. [62], pages 1157–1160.

[93] Jingren Zhou, Per-ke Larson, and Hicham G. Elmongui. Lazy maintenance of materialized views. In *Proc. of the 33rd International Conference on Very Large Data Bases (VLDB'07)*, pages 231–242, 2007.

[94] Jingren Zhou, Per-ke Larson, Jonathan Goldstein, and Luping Ding. Dynamic materialized views. In *Proc. of the 23rd International Conference on Data Engineering (ICDE'07)*, pages 526–535, 2007.