YOUTOPIA: A COMMUNITY DATABASE MANAGEMENT SYSTEM

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

Lucja Anna Kot

February 2010

YOUTOPIA: A COMMUNITY DATABASE MANAGEMENT SYSTEM

Lucja Anna Kot, Ph.D.

Cornell University 2010

This thesis introduces Youtopia, a system for collaborative management of relational data. In the age of Web 2.0, the sharing of relational data by communities is an increasingly important phenomenon. As a data management setting, it poses unique technical challenges which warrant dedicated solutions. We focus on two key aspects of Youtopia functionality: data cleanliness maintenance and handling interference between user tasks – or transactions – that access the same data.

We present a set of operations for maintaining data cleanliness in a collaborative manner. A unique feature of our operation set is the mechanism for enforcing tuple-generating dependencies (a formalism for constraint maintenance). In Youtopia, these are enforced through a process based on the classical chase that involves both automated and human-assisted steps.

Next, we examine the issue of transaction interference. We provide a suitable definition of serializability for the Youtopia transaction model and present an algorithm framework that can be used to enforce it. We illustrate our framework with an extended case study that includes experimental results from the implementation of our algorithms. Finally, we begin the investigation of concurrency control notions other than serializability. We present several isolation levels for transactions which are less restrictive than full serializability, but can be implemented with much more lightweight mechanisms and thus enable the system to achieve better throughput and latency in the processing of user operations.

BIOGRAPHICAL SKETCH

Łucja Anna Kot was born in Kraków, Poland. She completed her undergraduate education at the University of Auckland, New Zealand, earning both Bachelor of Science and Bachelor of Arts degrees in 2000. She began her graduate studies at the University of Pennsylvania, earning a Master of Arts degree in Romance Languages in 2002. She subsequently moved to Cornell University and changed her field of study to Computer Science; she is completing her Ph.D. degree in the subject in 2010.

ACKNOWLEDGEMENTS

I would like to thank my committee, Professors Christoph Koch, Johannes Gehrke, Dexter Kozen and Richard Shore, for their help and advice during my graduate studies and while working on this thesis. I would also like to thank all other faculty members who have been my teachers and mentors throughout my graduate and undergraduate studies.

Outside the purely academic setting, a great many people have been very supportive in ways large and small, and I would like to express my gratitude to them here. This includes all my officemates at Cornell, as well as other fellow graduate students, in computer science and other departments. Last but not least, thanks are also due to all my family for their help and support over the years. A particular thank you goes out to my husband Walker and my two cats Marcus and Lyta for helping me to stay sane through it all.

A portion of the material in this thesis is joint work with Christoph Koch. Moreover, an earlier version of some of the material from Chapters 1, 2 and 3 was previously published as: Łucja Kot and Christoph Koch, *Cooperative Update Exchange in the Youtopia System*, in PVLDB 2(1):193-204 (2009). This material is reused here in accordance with the relevant copyright policy under which authors retain the right to use all or part of their articles in their own future work.

This material is based upon work supported by the National Science Foundation under Grants IIS-0534404 and IIS-0911036 and by New York State Science Technology and Academic Research under agreement number C050061. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF) or of NYSTAR.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Collaborative database management systems

In the age of Web 2.0, the sharing and management of data by communities is ubiquitous. Groups of people share data for a wide variety of reasons, from entertainment or commercial activity to collaboration on scientific or artistic projects. The data involved is also highly varied, running the gamut from unstructured through semistructured to relational. Current systems used for data sharing are most often custom-built for a concrete scenario; as such, they exhibit significant diversity as well. To name only a few prominent solutions, Wiki software has proved very successful for community management of unstructured data; scientific portals such as BIRN [2] and GEON [4] allow scientists to pool their datasets; and an increasingly large number of vertical social networking sites include a custom, topic-specific database that is maintained by the site's members.

The Youtopia system is a general-purpose solution to enable community data sharing in arbitrary settings. Our initial focus within the Youtopia project has been on relational data; however, the ultimate goal is to include arbitrary data formats and manage the data in its full heterogeneity, as in Dataspaces [41].

Before introducing Youtopia in more detail, we begin by addressing a fundamental question. Is a custom-designed system for community data management truly necessary? After all, relational data could in principle be shared using existing technology, by having the community simply use a database. Of course, some additional functionality – beyond that of a traditional DBMS – would be desirable to communities, and providing it might pose a few practical challenges. For example, the DBMS would require an interface that is easy for non-experts to use. It would also have to incorporate policies and mechanisms for handling disagreement: Who should have administrative privileges on which tables? How should edit wars be handled? And so on. While certainly nontrivial, these issues seem in principle resolvable without recourse to radically new technology.

Our answer to the above motivational question is in the affirmative. While a normal database can certainly be used by a community, it is seldom the best-fit solution. This is because the community data sharing setting comes with unique features that create both challenges and opportunities for system design. As those features are often very different from those found in "classical" database use scenarios (such as business transaction processing), entirely new system designs are both possible and appropriate.

What is unique about the community setting? Without proposing to give an exhaustive list of differences, we briefly discuss several key aspects that are of particular importance.

First, a community DBMS should be *logically decentralized* as far as possible. That is, no functionality should require complete understanding by a human of all the data and/or metadata. As a database and a community grow, it becomes more and more difficult for individual users (or very tightly cooperating groups) to maintain such in-depth knowledge, and bottlenecks inevitably arise. It is much better to redesign all functionality so that any tasks carried out by humans only require a small amount of local domain knowledge.

Redesigning a DBMS to achieve logical decentralization can be highly nontrivial, for many reasons. Beyond the obvious ones, there is the issue that the global knowledge requirements may not be explicit in the system. For example, they may come in the form of global assumptions or constraints about the data or metadata that must be maintained as the database evolves. Even if these constraints are not directly used by humans (because, for example, they serve as prerequisites for certain *automated* processes to run), their *enforcement* as the data and schema changes may require global domain knowledge from a human. We will see an example of such a constraint in Section 1.3, when we discuss update exchange.

In a logically decentralized community DBMS, all tasks involving the data and metadata will be be carried out collaboratively by groups of users. If no single user has enough domain knowledge to complete a task, the system must allow people to pool their limited understanding by each contributing as they are able. This means the tasks must be possible to perform on a *best-effort basis*, with the "minimum meaningful unit of contribution" being as small as possible. Of course, there is no reason why all the principals collaborating on a task need to be human: algorithms can be used to assist users in the process as well.

If all work on the data is carried out on a best-effort basis, the system will inevitably contain – at any given point in time – many partially-completed tasks, and consequently data (and metadata) that is incomplete, inconsistent or otherwise dirty. Dirty data will be found in a community DBMS for other reasons too, for example due to bulk-loading of a table from a dirty external source. The system must handle such data gracefully for the purposes of query answering and permitting other tasks to proceed. It should also come with a comprehensive suite of tools for data cleaning; users must, of course, be able to perform this cleaning on a best-effort basis.

Another issue that cannot be avoided in a community DBMS is disagreement management. Users will inevitably disagree on data, and mechanisms must be in place for handling this. Whether the disagreement is handled “inside” the system (by, say, allowing multiple versions of tuples), or “externally” (through an arbitration system as in Wikipedia), the system must provide support for disagreement-related tasks. At the very minimum, comprehensive logs of user operations must be available and easy to query, so that interested parties can understand the nature and scope of the disagreement. Indeed, such logs have other applications too: for example, they can be used to obtain provenance information to determine the reliability of the data in cases where trust is an issue.

The last challenge we mention here is making optimal use of human attention to improve data quality. In a community DBMS, we will frequently be very lucky in that we will have an active group of users who are motivated to improve data quality, as in Wikipedia. However, this human attention should not be squandered, and it is helpful if the system is equipped to exploit it as well as possible. For example, if automated cleaning rules can be inferred from user cleaning behavior, they would likely greatly reduce the amount of drudge work needed from the humans and thus improve the system tremendously.

In the next section of this chapter, we present two use scenarios for Youtopia; they will serve as a source of running examples throughout this thesis. In the following section, we introduce the two particular aspects of classical DBMS functionality which we have been working on adapting to the community setting: the maintenance of data cleanliness and concurrency control. Finally, Section 1.4 contains a detailed list of the specific technical contributions we make in this thesis.

1.2 Use scenarios: the travel website and the library

Our first use scenario is a large, public, Web-based community repository containing travel information. Modern travel-related websites such as Wikitravel frequently contain a large amount of structured data: calendars of events, train or bus schedules, or just geopolitical information that is inherently structured. For example, the English-language page on Wikitravel devoted to Europe includes a large table detailing, for each European country, whether it belongs to the European Union (and if so, since when), whether it uses the Euro and whether it belongs to the Schengen zone. [12]

While Wikitravel and similar websites such as Tripit [10] are currently useful, any travel site could provide a significantly richer user experience by making better use of structured data. For example, it could allow users to perform information discovery queries ("What are the biggest cities in Europe, across all countries?"). It could also allow them to create their own private tables that could be shared with friends - for example, a user might create a table to keep track of the cities in the world that they have visited, together with trip dates. It could even provide an API allowing external developers to create applications that use the structured data in interesting ways. For instance, it would be useful to have a route planning program that takes as input a table of the user's desired destinations, and uses another table containing train schedules to produce a suggested day-by-day itinerary.

In practice, the users of such a travel data repository would also wish to share less-structured data such as text and photos. Thus, a complete end-to-end solution would be a composite system that includes both a Youtopia repository and other components such as a Wiki. Such multi-component systems are used today to cater to particular vertical communities; for example, Assembla [1] is an online platform for collaborative software development, and includes a Subversion repository, but also provides a Wiki and a task assignment system. In the case of the travel website, an ideal solution would seamlessly integrate the structured and unstructured data and tools. Initially, however, we restrict ourselves to thinking about a travel database that contains only structured data.

In our second scenario, Youtopia is used by the owners and patrons of a small community library. The owners use it to maintain administrative data relevant to the running of the library, such as book holdings, wishlists and event calendars; the patrons can create tables of their own on any topics of interest to the community, such as private reading lists, book reviews, and so on. Thus, a portion of the repository is very open and functions much like the travel website. However, the functionality associated with the administrative tables used is much closer to that found in a traditional OLTP database system.

We note that there is currently both a real need and a lack of solutions for a system that would provide RDBMS-like functionality while remaining lightweight and easy to use. There are many settings where a set of spreadsheets is not adequate, but where a full-fledged RDBMS is too expensive or difficult to set up. Many grass-roots communities have needs that fall exactly into this intermediate space, and there is a great level of interest in systems to address these needs. This interest is evident, for example, in many posts found in the Google Fusion Tables discussion group [7]. Youtopia, which is by design a lightweight but powerful system suited to settings where users have limited understanding of both the data and database technology, is uniquely positioned to fill this niche.

1.3 Working with data collaboratively

This section discusses data cleaning and concurrency control in community DBMSs, and introduces the unique challenges posed by their redesign for logically decentralized, best-effort operation.

1.3.1 Data cleaning

In any database, a key goal is maintaining data that is as clean and logically consistent as possible, in the face of data modifications both small (individual tuple inserts or deletes) and large (bulk loading of data from external sources). Some data cleaning can be done by hand by humans; indeed, in community settings, users are particularly likely to be willing and able to perform basic cleaning operations. For the same reasons that Wikipedia attracts an army of people willing to correct and improve articles, it is realistic to expect that our travel website's users will be willing and able to perform simple operations like duplicate identification and removal. Of course, if the repository is sufficiently large and the rate of changes to the data is high, humans will not be able to keep up. For this and other reasons, it is definitely worthwhile to leverage data cleaning algorithms to assist in the process. In the best of all worlds, most of the cleaning work would be offloaded to an algorithm, with humans only stepping in occasionally at crucial points. This hybrid approach is actively being investigated by researchers today [43, 24].

Besides duplicate handling, another aspect of cleaning where the system can help is the maintenance of data consistency with respect to logical constraints. The simplest example of this is maintaining the consistency of copies, near-copies or otherwise overlapping data. Overlapping data arises naturally in the community setting, and this is not always a pathological phenomenon due to inadequate normalization or similar issues. Users may, for example, create materialized views for data presentation purposes. Or they may disagree on a portion of the data while agreeing on another, a common occurrence in scientific communities [60]. This requires the maintenance of two (materialized or virtual) versions of the relevant data that agree on the "uncontroversial" portion. Clearly, any updates to duplicated, shared and/or overlapping data should be propagated in a way that ensures consistency. If a tuple is inserted into one copy of a table, the same tuple must be inserted into all others so that they remain true copies.

A slightly different, but related, kind of logical constraints are those that establish a relationship between some data and other data that is (partially) derivative of it, where the derivation process is more complex than making a simple copy. Consider the example travel database shown in Figure 1.1. The `R` table contains reviews of attraction tours in New York State; suppose we would like to ensure that all attractions in New York State that have available tours are indeed reviewed, or at least get the chance to be reviewed by becoming visible in the `R` table. This can be accomplished through a *tuple-generating dependency (tgd)*. Concretely, the tgd σ_3 allows us to express and enforce the constraint above, as explained in the following example.

Example 1.3.1. *Suppose company `ABC Tours` starts running tours to Niagara Falls and the tuple `T(Niagara Falls, ABC Tours)` is added. The tgd will cause the new tuple `R(Niagara Falls, ABC Tours,` x_3`)` to be inserted by the system. The x_3 is a* labelled null *or* variable *which indicates that some review for the tour should exist, but is unknown to the system. The review may subsequently be filled in manually by a user.*

C (City)

city
Ithaca
Syracuse

S (Suggested Airport)

code	location	city served
SYR	Syracuse	Syracuse
SYR	Syracuse	Ithaca

A (Attraction)

location	name
Geneva	Geneva Winery
Niagara Falls	Niagara Falls

T (Tours)

attraction	company	tour start
Geneva Winery	XYZ	Syracuse
Niagara Falls	x_1	Toronto

R (Tour Reviews)

company	attraction	review
XYZ	Geneva Winery	Great!
x_1	Niagara Falls	x_2

V (Conventions)

city	convention
Syracuse	Science Conf

E (Excursion Ideas)

convention	attraction
Science Conf	Geneva Winery

$$\sigma_1 : \mathtt{C}(c) \rightarrow \exists a, c' \ \mathtt{S}(a, c', c)$$
$$\sigma_2 : \mathtt{S}(a, c', c) \rightarrow \mathtt{C}(c') \wedge \mathtt{C}(c)$$
$$\sigma_3 : \mathtt{A}(l, n) \wedge \mathtt{T}(n, c, c') \rightarrow \exists r \ \mathtt{R}(c, n, r)$$
$$\sigma_4 : \mathtt{V}(c', x) \wedge \mathtt{T}(n, c, c') \rightarrow \mathtt{E}(x, n)$$

Figure 1.1: Portion of the travel database

The process of this propagation of changes is known as the (tgd) *chase* [14, 50, 25] – a simple mechanism for constraint maintenance in which the corrective operations required are relatively easy to determine and perform.

Tuple-generating dependencies and equivalent constraints such as GLAV mappings [49] and conjunctive inclusion dependencies [47] are frequently encountered in data integration [39, 30, 42, 45, 63]. Their ubiquity points to the fact that they are a very powerful formalism, applicable in a variety of subject domains. They can also handle the simpler constraints described before - copy consistency maintenance and view maintenance.

The other tgds in Figure 1.1 enforce additional constraints, as follows. σ_1 states that every city has a recommended airport. Under σ_2, every airport is located in a city and serves a city. Because of σ_4, convention attendees can receive recommendations for day trips based on the convention venue and available tours.

Since tgds are such a powerful and versatile tool, we would like Youtopia users to be able to create them and add them to the system as easily as they add new tables. There are, however, several reasons why this is hard. First, it is not always trivial for a user to specify a tgd that correctly reflects the intuitive constraint they have in mind. Still, the issue of tgd creation has been addressed in some existing work [63, 56] and we are building on these solutions to set up an infrastructure to facilitate this process. Notably, Youtopia allows users to cooperate and pool their understanding to set up and refine tgds. Dependency creation can also be made easier by the presence of subdomain-specific summary views: knowledgeable users can define such views which capture in their schema the essence of the subdomain. Much as portals and topic lists in Wikipedia can guide contributors in the categorization of their articles, such views can guide table owners in the formulation of their tgds.

The other problem with allowing ad-hoc tgd creation by users is more serious. If the dependencies happen to have cycles among themselves, as do our σ_1 and σ_2, it is sometimes possible for the system to start performing a *nonterminating chase* that involves an infinite cascade of inserts.

Example 1.3.2. *Suppose JFK airport is added as a suggested access airport for Ithaca. The tuple* `S(JFK, NYC, Ithaca)` *is added to the database. To satisfy σ_2, we need to add tuple* `C(NYC)`*. This in turn causes a violation of σ_1, which is repaired by inserting* S(x_3, x_4, NYC)*. This new insert causes a violation of σ_2, requiring the insertion of* C(x_4)*, and so on.*

Such cyclical firing of rules is a well-known problem; current research has handled it by placing a global acyclicity restriction on the dependencies [47, 45, 39, 30]. However, in a collaborative setting, such a restriction is unrealistic and undesirable, for at least two reasons. First, it violates our logical decentralization desideratum for system design. As the number of tgds grows, any cycles that are created "by accident" are increasingly likely to be large and complex. A user who is warned by the system that the tgd they want to add would create a cycle may not have the global knowledge to understand, let alone resolve, this problem. Second, cyclical dependencies are not always a pathological phenomenon, so disallowing them can prevent users from expressing meaningful constraints. σ_1 and σ_2 above are an example of this; we also give another example from a different subject domain.

Example 1.3.3. *Suppose we wish to maintain information about people's ancestry in a genealogical database such as Geni [3]. It is natural to do this with three tables,* `Person`*,* `FatherOf` *and* `MotherOf`*, and tgds to connect these three together. First, every person has a father and a mother, so we set up* *a tgd* $\texttt{Person}(x) \rightarrow \exists y,z\ \texttt{FatherOf}(y,x) \wedge \texttt{MotherOf}(z,x)$*. Also, every father and mother is a person, so we add the tgds* $\texttt{FatherOf}(x,y) \rightarrow \texttt{Person}(x)$ *and* $\texttt{MotherOf}(x,y) \rightarrow \texttt{Person}(x)$*. These tgds are cyclic; inserting, for example,* $\texttt{Person}(\texttt{John})$ *into an empty database will once again lead to an infinite cascade of insertions.*

Here, the cycle is clearly not pathological; the existence of an unbounded chain of ancestors is actually consistent with the conceptual model that the users had in mind when they set up the dependencies. After all, anyone should always be able to add more information about further and further ancestors as they research their family's history. However, the chase enforcement mechanism for the tgds is required by definition to materialize the entire unbounded chain of ancestors; this of course is a nonterminating process.

In Chapter 2 we present a different enforcement mechanism for tgds that provides a solution to the nonterminating chase problem. We also explain how tgds and our enforcement mechanism can function as an integral part of a broader set of data cleaning operations.

1.3.2 Concurrency control

As users work with data to achieve particular goals, they will naturally perform tasks that involve multiple steps and operations. Even in a hybrid system where both human and automated principals collaborate, many of the steps in a task will be carried out by humans. Because of this, the tasks will be "long-running" in system terms, as the delays between steps are likely to be substantial: in the best case, on the order of minutes, in the worst, of days or more. We now give two examples of long-running tasks.

Example 1.3.4 (Community library book order)**.** *This example is set in the community library scenario and involves library employees placing a book order. The task might proceed as follows:*

- *Kate starts the task and creates the* `Order` *table to contain details of the books to be ordered.*
- *Jim queries the* `Book Wishlist` *table to determine the 15 most frequently wished-for books and inserts them into the* `Order` *table.*
- *Jane organizes a series of talks by visiting authors, and knows who is visiting next year based on information in the* `Visitors` *table. She selects suitable books by the visiting authors (e.g. by finding the 2 most highly ranked ones for each author), and if the books are not already either in current library holdings or in the* `Order` *table, they get added too.*
- *Tom inserts more books into* `Order` *based on another query (e.g. textbooks used in the local community college this year)*
- *Daisy notices that given the state of the* `Order` *table, there is still enough money in the budget for a new edition of an important reference book , so she adds it too.*
- *Kate places the order based on the current state of the* `Order` *table and concludes the task.*

Example 1.3.5 (Planning a trip)**.** *Our second example involves the travel database; here, Alice and Bob plan a trip to France together. They will be traveling together for the first two weeks, but will split up for the third and final week.*

- *Alice and Bob begin by creating a table with their arrival and departure information, budget, etc. and begin to plan the first two weeks.*
- *Alice wants to visit some castles, so she queries for the 5 most popular castles in France and creates a small table containing information about them.*
- *Bob is interested in music festivals, so he creates a table devoted to these.*
- *Alice and Bob run a Youtopia application that plans a trip for them based on the castles and festivals they want to see, taking into account available transportation options, their budget, etc.*
- *The application creates a table with a day-by-day itinerary.*
- *Alice and Bob now plan their separate trips for the third week; each creates their own tables, itineraries, etc, relating to their activities during that week.*

As we see in the the trip planning example, it is possible for tasks to nest hierarchically within others; Alice's and Bob's separate trip plans for the third week should probably be modeled as nested subtasks of the main trip one.

Collaborative tasks are then a sequence of logically related operations, where the notion of "logically related" is defined by the users themselves. As such, they are somewhat similar to waves in Google Wave [8]. However, Google Waves are documents themselves, while a collaborative database task consists of the queries and writes performed on the data, not the data itself; as such, it is purely a meta-level concept. Indeed, we can imagine using an already-executed task to generate a template that could be reused on different data. For example, we could create a task template that would automatically generate a sample trip itinerary, given some basic input parameters such as destination country, budget constraints, specific interests etc.

Of course, none of these tasks are carried out in isolation, on an otherwise unused system. Interference between different tasks can and will arise, as can be seen in the following examples.

Example 1.3.6. *Return to the book order scenario in Example 1.3.4. After Jane inserts her books based on who will be visiting, one more visitor is scheduled, and the tuple with their name is inserted into the* `Visitors` *table. So, Jane may need/want to order books by that author too. This of course may affect other events downstream, such as Daisy's ability to use up the rest of the budget on an extra book.*

Example 1.3.7. *This example continues the trip planning scenario from Example 1.3.5, with the main trip task and two subtasks for the third week. Suppose Bob realizes the end date for the trip needs to be moved forward by two days, because he has to be back earlier. So he changes that information in a table in the main task for the trip. That change affects the itineraries in the subtasks, as they were computed with the old trip end date in mind.*

Example 1.3.8. *Again consider the trip from Example 1.3.5. Suppose there is another task in the system that continuously gathers information about all the trips in the system, calculating – say – general statistics about most popular attractions, most common trip lengths and budgets, etc. Now Alice and Bob decide they need to cancel their trip completely. The statistics-gathering task has read information about their trip, however, and it is therefore making statistical computations based – in part – on inaccurate information.*

It is relatively clear that interference can be problematic; indeed, in Examples 1.3.6 and 1.3.7, it is probably desirable for some corrective action to be taken, manually or by the system. In the last example, however, this is actually less clear

– the statistics-gathering task may not require equally precise guarantees. Statistics based on somewhat out-of-date information may be acceptable, particularly if correcting the problem would require a heavyweight operation like a restart of the entire statistics-gathering process.

Dealing with task interference in a collaborative setting is obviously related to the problem of maintaining isolation between traditional OLTP transactions. Because of this, we henceforth use the term *transaction* to refer to the collaborative tasks as well. However, our transactions have several important “non-traditional” features which mean that existing OLTP concurrency control solutions are not well-suited to them. Because of this, it is desirable to provide a dedicated concurrency control protocol for them – another argument for a dedicated collaborative DBMS.

How, more precisely, is concurrency control different in a collaborative DBMS? To answer this question, we need to clarify and (somewhat) formalize our model of a transaction. It is simply a sequence of reads and writes, where the reads are specified intensionally (with queries), and the writes are specified extensionally. Because of the user interaction element, arbitrarily long delays may occur between two consecutive steps of a transaction. The last operation of a transaction may be a *commit* operation, but this is not required. When a commit operation does occur, the desired effect is that of a standard OLTP commit, i.e., durability for this transaction is desired. An example of a transaction that likely does need to commit is the one in Example 1.3.4. On the other hand, the one in Example 1.3.5 probably does not need to.

When transactions are actually running in the system, they can be in one of three states – executing, waiting for user input and *ready*. The first two are self-explanatory. In the ready state, a step for this transaction is available (in the sense of being known to the system) and is only pending scheduling to run.

The following collaborative DBMS features pose unique challenges in implementing a concurrency control scheme:

- no transaction in the ready state may be required by the system to wait for an indefinitely long period of time. Thus any concurrency control protocol must be non-blocking, in the sense that no operation can be forced to wait until another transaction's operation(s) complete. This is of course because the other transaction may be subject to arbitrary delays.
- any transaction that has not performed an explicit commit may abort at any time due to an explicit user request.
- all other aborts (i.e. cascading ones required to maintain appropriate concurrency control behavior) are extremely undesirable and should be avoided.

On the other hand, the two following differences from the OLTP scenario can make the design of concurrency control algorithms easier:

- not all transactions end with a commit operation; in fact, the majority will not, and can thus be rolled back if necessary. Of course, aborts are very undesirable, as mentioned above, but they are seldom actually impossible.
- it is much more acceptable to relax requirements for data consistency (and consequently, for transaction isolation).

Within the above constraints, there are several choices on what to enforce and how to enforce it, and the exact level of enforcement is likely to depend on the specific setting and indeed the specific transaction (as is evident in the difference between Examples 1.3.7 and 1.3.8). This suggests that the best solution is a concurrency control framework providing a variety of options or *levels* of enforcement, allowing users and/or system deployers to choose what is most suitable for them. We begin the investigation of the design space for such levels in Chapters 3 and 4.

1.4 Contributions and outline of this thesis

In this section, we present the technical contributions of this thesis in the form of a chapter-by-chapter outline.

First, we show how the data cleanliness maintenance process from Section 1.3 can be redesigned for a best-effort collaborative setting. In Chapter 2 we introduce a set of basic operations for data cleaning, including explanations of how they interact with each other. We also present a new mechanism for tuple-generating dependency enforcement which has several advantages over the traditional chase. First, it comes much closer to the intuitive enforcement model that is implicit in tgd semantics. Second, it allows us to remove the acyclicity constraint on dependencies. Finally, it exemplifies a novel hybrid computation model for constraint enforcement, where as much work as possible is carried out automatically, but users occasionally step in and assist the algorithm with their domain knowledge.

In Chapter 3, we begin the investigation of transaction interference. We introduce a notion of serializability that is appropriate for the collaborative DBMS setting and relate it to classical notions of serializability found in the literature. We then give a protocol for enforcing serializability, including an algorithm framework. We present a case study for serializability enforcement in a restricted setting, where transactions may only use a limited subset of operations from those discussed in

Chapter 2. Our presentation includes specific algorithms and experimental results.

In Chapter 4, we begin a broader investigation of concurrency control enforcement options for transactions. We discuss in depth the unique constraints, desiderata and tradeoffs associated with handling transaction interference in community data management. We also present several *isolation levels* for transactions which are less restrictive than serializability, and can thus be enforced with more lightweight mechanisms (albeit at the cost of some loss in data consistency).

Chapter 5 contains a brief overview of the literature relevant to community database management, with a focus on the work that provides specific technical background for the research presented here.

CHAPTER 2

DATA CLEANING IN YOUTOPIA

In this chapter, we present the collaborative data cleaning functionality provided by Youtopia. Our tools allow users to perform operations that improve data quality with respect to two concepts: *duplicate* tuples and *corresponding* tuples from different tables. As we show in the first two sections of this chapter, these concepts have very natural formalizations which are associated with an intuitive set of manipulation operations. However, in several cases, there are multiple options as to the exact definition of the operations and there is room for design decisions to be made. We indicate these decision opportunities as they arise in our discussion. The last section of this chapter presents a concrete set of operations chosen from the available alternatives. Our operation set is designed to provide a large amount of data cleaning functionality, and allows for the participation of both humans and algorithms in the cleaning process.

We make a note here about queries in Youtopia. In what follows, we frequently talk about principals, both human and automatic, querying the database. Algorithms can of course pose arbitrary queries – in SQL or in any other language – as they wish. We assume that the human users can also pose simple SQL queries, including at least select-project-join queries and aggregation. Making the latter happen with non-expert users is of course a nontrivial interface design problem, but as can be seen in Google Base [5] and Fusion Tables [6], form-based interfaces can allow users to construct quite sophisticated queries in a relatively intuitive way. Thus, our last assumption is not an unrealistic one.

	name	state	population
C (US Cities)	Wilmington	Delaware	73,000
	Wilmington	North Carolina	100,000
	Wilmington	x_1	90,000

Figure 2.1: Example of duplicate tuples

2.1 Duplicate tuples

Consider the example table in Figure 2.1, which might be encountered in our travel website scenario. It contains information about US cities, and includes three tuples referring to cities named Wilmington. On inspection, one might come to suspect that the third tuple may be a duplicate of one of the first two. This is not a difficult observation to make; however, determining which tuple is being duplicated may be much harder.

Youtopia provides tools that allow users to collaboratively resolve the problem of the third tuple. It is possible for a user to mark the appropriate pairs of tuples as duplicate candidates, without going further. Another user with more domain knowledge can subsequently decide which tuples are indeed duplicates and remove the redundancy. In Youtopia, we allow users to perform the following operations related to duplicate management:

- *mark* two tuples as potential duplicates
- reverse the above, i.e. *unmark* two tuples as potential duplicates
- *merge* two tuples which are marked as duplicate candidates
- reverse the above, i.e. *split* a tuple into two new tuples (not marked as duplicate candidates)

These four operations are simple and should be easy to understand for all users. Interestingly, they have counterparts for unstructured data in the world of Wikipedia, where pairs of articles may be flagged as candidates for a merge and subsequently merged. Both the marking and merging are, of course, reversible [11]. The operations are also simple to implement and require only minimal additional metadata. For every table, we need to maintain a binary relation **Dup** specifying which pairs of tuples are marked as potential duplicates. Note that mathematically, this relation is irreflexive, symmetric, and not transitive; in Figure 2.1, although the third tuple is suspected of being a duplicate of one of the first two, it does not automatically follow that the first two tuples may themselves be duplicates.

There is a design decision to be made regarding the extent to which information in the relation **Dup** is visible to the user. For data browsing, it is certainly reasonable and indeed desirable to expose the potential duplicate information, either right away or on demand. For queries, however, the issue is less clear. The simplest choice is to ignore it, answering queries on the database in a way that disregards this information. For example, the answer to the query `SELECT COUNT(*) FROM C` would be 3. A more sophisticated solution would be to take **Dup** information into account for at least some queries – indeed, aggregation queries such as the one just presented would be an obvious choice. It may be helpful to a user to learn that there are at least two mid-sized cities called Wilmington in the US, and possibly three, but no more. Developing suitable query semantics in such a setting is future work. This type of extension is not trivial and requires serious thought since – as we will see – all our data cleaning operations, including queries, interact with each other, and the semantics of one cannot be extended or modified without considering this change's impact on the others.

2.2 Corresponding tuples

2.2.1 Table-level correspondences

In addition to managing duplicate (and potentially duplicate) tuples, Youtopia users can perform operations to deal with tuple overlap and correspondence. At the underlying mathematical level, Youtopia treats both of these concepts identically, despite the fact that they appear different at first glance. In common parlance, the term *overlap* is most likely to refer to data duplication caused by the presence of copies of tables and/or derived materialized views. When users talk about *correspondence*, on the other hand, they are more likely to mean inter-table relationships such as the one in Figure 2.2. This figure presents two tables from our library scenario. The first contains information about biographies, the second about famous people. We expect that every person who is the subject of a biography is relatively famous, and as such, more data on them is to be found in the `Famous People` table. We saw more examples of such correspondences in the database in Figure 1.1.

The reason we treat overlap and correspondence as a single phenomenon is that both can be expressed with the same mathematical formalism, namely that of tuple-generating dependencies or tgds [30, 60]. Formally, a tuple-generating dependency has the form

$$\Phi(\overline{x}, \overline{y}) \rightarrow \exists \overline{z} \Psi(\overline{x}, \overline{z})$$

where Φ is a conjunction of relational atoms over the sets of variables and constants $\overline{x}$ and $\overline{y}$ (each of these may contain both variables and constants), while Ψ is a similar conjunction of relational atoms over $\overline{x}$ and $\overline{z}$. The free variables are understood to be universally quantified. σ_5 in Figure 2.2 is an example of a tgd and

B (Biographies)

ISBN	title	subject_lastname
123	Roosevelt	Roosevelt
456	Roosevelt	Roosevelt

P (Famous People)

firstname	lastname	DOB
Eleanor	Roosevelt	10/11/1884
Franklin D.	Roosevelt	1/30/1882
Theodore	Roosevelt	10/27/1858

$$\sigma_5 : \mathtt{B}(i, t, l) \rightarrow \exists f, d\ \mathtt{P}(f, l, d)$$

Figure 2.2: Example of data with correspondences

expresses the correspondence between the two tables which we previously explained in English. In what follows, we will also sometimes use the term *mappings* to refer to tgds.

How, precisely, do tuple-generating dependencies express table-level overlap and/or correspondence? In some high-level sense, they are logical formulas that are satisfied by the database iff the overlap or correspondence exists. To make this more precise, we need to define what it means for a formula to be satisfied over a Youtopia database. This is not trivial; for example, our definition must take into account the fact that Youtopia databases may contain variables (labeled nulls) in addition to constants.

For reasons which will become clear shortly, we first give the definition of tgd satisfaction in a restricted case, where both the left-hand side (LHS) and right-hand side (RHS) of the tgd contain only one atom. This is actually a very common and important subcase, as for example the fact of being a copy or derived view can be expressed with such mappings. If R and S have three attributes and are both copies of each other, this can be expressed with the mappings $\mathtt{R}(x, y, z) \rightarrow \mathtt{S}(x, y, z)$

and $\mathtt{S}(x, y, z) \rightarrow \mathtt{R}(x, y, z)$.

Definition 2.2.1 (Database). *A database D is a finite set of finite relations. The relations contain attribute values which are either constants from a finite set $Const$ or labeled nulls from a finite set Var.*

Definition 2.2.2 (Tgd satisfaction, join-free case). *Consider mappings μ from a set X of variables and constants (which appear in tgds) to $Var \cup Const$. Restrict the mappings so that $\mu(c) = c$ for every $c \in Const$. Given such a μ, we can extend it to a mapping from relational atoms to relational atoms, by taking for a given relation $\mathtt{R}$ of arity n, $\mu(\mathtt{R}(x_1, x_2, \cdots x_n) = \mathtt{R}(\mu(x_1), \mu(x_2), \cdots \mu(x_n))$.*

Let σ be a tgd with a single atom on the left-hand side (LHS) and a single atom on the right-hand side (RHS), so that it has the form

$$\mathtt{A}(\overline{x}, \overline{y}) \rightarrow \exists \overline{z}\ \mathtt{B}(\overline{x}, \overline{z})$$

for some tables $\mathtt{A}$ and $\mathtt{B}$. σ holds on D if, for every mapping μ as presented above defined on $\overline{x} \cup \overline{y}$, whenever $\mu(\mathtt{A}(\overline{x}, \overline{y})) \in D$, μ can be extended to a mapping μ' from $\overline{x} \cup \overline{y} \cup \overline{z}$ to $Var \cup Const$, with the property that $\mu'(\mathtt{B}(\overline{x}, \overline{z})) \in D$.

We note that Definition 2.2.2 is closely related to the corresponding one in [30]. However, our notion of tgd satisfaction will diverge from theirs in the multi-atom case.

2.2.2 Tuple-level correspondences

Before we deal with the multi-atom case, we discuss a second kind of correspondence that is meaningful and defined in Youtopia – tuple-level correspondence.

Return to the example database in Figure 2.2. As previously explained, σ_5 expresses the intuitive table-level correspondence we have previously discussed, and it holds on this database according to Definition 2.2.2. There is, however, additional information about the correspondence which users might be able to supply and which they might be interested in querying. The database contains tuples about two biographies of people named Roosevelt. Which Roosevelt(s) are they biographies of? This information would be very useful to someone interested, say, only in books about Theodore Roosevelt. It is also information that might be easy for users to supply – if the biography tuples are normally inserted one at a time, by a person holding the physical book, it is very easy for that person to specify the appropriate Roosevelt each book at insertion time.

Such tuple-level correspondences are easy to model using a mathematical relation defined on tuples from different tables (unlike **Dup**, which related tuples from the *same* table). In the example in Figure 2.2, we would define a binary relation on tuples from tables `B` and `P` to contain information about the per-tuple correspondences.

There is the question of whether this new relation should be unrestricted, or – for example – constrained so that all correspondences are many-to-one. The latter choice is the one we make in our current system. This decision is not overly restrictive, as many-to-one correspondences are by far the most natural for users to express and work with. Moreover, this design choice makes it very easy to represent the correspondence information in a way that allows for fast update and retrieval, even in queries that involve a large amount the data. We explain how this representation works using an example.

Consider Figure 2.3, which contains the same data as in Figure 2.2, but has been

B (Biographies)

ISBN	title	subject_lastname	person_key
123	Roosevelt	Roosevelt	2
456	Roosevelt	Roosevelt	0

P (Famous People)

firstname	lastname	DOB	key
Eleanor	Roosevelt	10/11/1884	1
Franklin D.	Roosevelt	1/30/1882	2
Theodore	Roosevelt	10/27/1858	3

$$\sigma_5 : \texttt{B}(i, t, l) \rightarrow \exists f, d\ \texttt{P}(f, l, d)$$

Figure 2.3: Example of explicitly specified tuple-level correspondences

augmented with metadata to represent tuple-level correspondences. The P table has been extended with an additional field which is a primary key; the additional attribute in B is a foreign key referencing that primary key. This should not be surprising, as σ_5 is essentially specifying an inclusion dependency between the two tables. There is one extra feature to our representation, which is the special value 0 in the second tuple of B. This is a *wildcard* value indicating that the corresponding tuple from B is unknown.

Note that the information we have represented about the tuple-level correspondences is still in a very strong sense related to σ_5 and its satisfaction on the database. Indeed, it gives us information beyond the fact that σ_5 holds on the database - it specifies *how* and *why* it holds. We formalize this intuition in the notion of *explicit witnesses.*

Definition 2.2.3 (Explicit tgd satisfaction witness, join-free case)**.** *Let σ be as in Definition 2.2.2. A* witness *to the satisfaction of σ is any pair of tuples $t_1 = \texttt{A}(\overline{w_1})$ and $t_2 = \texttt{B}(\overline{w_1})$ such that there exists a mapping μ from $\overline{x} \cup \overline{y} \cup \overline{z}$ to $\overline{w_1} \cup \overline{w_1}$ with the property that $\mu(c) = c$ for all $c \in Const$. An* explicit *witness to the satisfaction* *of σ is a witness to its satisfaction that has explicitly been marked as such by a human or algorithm.*

Now, what should happen when the two tables on Figure 2.3 are in fact joined on the last name of the biography subject? Intuitively, any available tuple-level correspondence information should be taken into account. If no such information is available, as in the case of the second `B` tuple, all *potentially* corresponding tuples from `R` should be included in the join result. We formalize this in the notion of a *best-effort join*. (The join is best-effort because it always returns the most accurate information about tuple-level correspondences that is available).

Definition 2.2.4 (Best-effort join with respect to a join-free tgd)**.** *Let σ be as before; we define the best-effort join of `A` and `B` with respect to σ. A relational join is a subset of the Cartesian product of a set of tables; here, for the sake of notational clarity, we give our definition as a decision criterion for determining whether a pair of tuples $t_1 \in$ `A` and $t_2 \in$ `B` should be included in the join result. Let $t_1 \in A$ be given. For this t_1, we include the following $t_2 \in B$:*

- *if t_1 is part of an explicit witness for σ with some $t_2 \in B$, then include t_2 only*
- *otherwise, include all $t_2 \in B$ that join – in the normal relational sense – with t_1. The set of attributes to be used for the join is determined by the position of the shared variables $\overline{x}$ in σ, in the obvious way.*

We note that although the representation mechanism used for tuple-level correspondences in Figure 2.3 requires all correspondences to be many-to-one, this is *not* a restriction present in Definitions 2.2.3 and 2.2.4. If desired, therefore, it is

```
SELECT * FROM B, P WHERE
            (B.person_key = P.key) OR
            (B.person_key = 0 AND B.subject_lastname = P.lastname)
```

Figure 2.4: Example SQL query for computing a best-effort join

possible to work with many-to-many correspondences, although a compact representation may be harder to achieve. However, if the correspondences *are* restricted to be many-to-one and our representation *is* used, then the best-effort join has the additional advantage of being extremely easy to compute. For example, the join we have been discussing between the tables in Figure 2.3 can be obtained with the SQL command in Figure 2.4.

We also note that the best-effort join with respect to a tgd is defined and possible to compute even on databases that do not satisfy the tgd in the first place. This is important, since, as we shall see, we will in general end up working with such "dirty" databases – either because some violations of tgds are new and have not yet been repaired, or because some principal in the system has decided not to repair them for one reason or another.

At this point, we are finally in a position to discuss tgds with more than one atom per side. The reason we were restricted to the single-atom case until now is that multi-atom tgds involve joins between relations in a fundamental way, and the join operator used – normal or best-effort – affects the definition of tgd satisfaction.

Suppose that the users of the database in Figure 2.3 now wish to set up a dependency similar to σ_3 in Figure 1.1, allowing for biographies to be reviewed. They create a table `R` to hold the reviews, with attributes $title$, $subject\_firstname$, *subject_lastname* and *review*, and create a tgd σ_6:

$$\mathtt{B}(i,t,l) \wedge \mathtt{P(f,l,d)} \rightarrow \exists r\ \mathtt{R}(t,f,l,r)$$

Now that the tgd is in place, how should the R table be populated to satisfy it? The dependency includes a conjunctive query on the LHS that represents a join between B and P; it is essentially stating that R is a projection of that join, extended with an additional *review* field. The question, of course, is which join should be used. In this case, it makes good sense to use the best-effort join with respect to σ_5. For other dependencies and other cases, this need not be true. The choice of join to be used is thus a property of the tgd itself – it is an additional statement about its semantics which it is reasonable to expect the user to specify when they create it. Formally, every instance of a $\wedge$ operator must come with a specification of the operator to be used when evaluating the join implied by the conjunctive formula.

Once the above is understood, it is possible to extend our definitions of tgd satisfaction, witness, and explicit witness, to the case of multi-atom tgds. It is also possible to extend our relational representation of tuple-level correspondences (using foreign key-like constructs augmented with wildcard values) to these tgds. Neither the mathematical nor representational extensions are given here; they are conceptually straightforward, yet require substantial notational machinery to develop fully. In the real world, the vast majority of tgds will involve one or at most two atoms per side (a three-way join is not trivial for most people to understand, let alone use in a tgd). Of those tgds that involve two atoms, the overwhelming majority will have a clear intended join semantics: the two relations being joined will either have a single tgd connecting them (most likely, an inclusion dependency), or have no applicable tgds and then the normal relational join will be necessary. Thus, in practice, and in the remainder of this thesis, we will assume that the join semantics are given as part of the specification of each tgd, and we will never mention them explicitly. We will also use the normal notation $\wedge$ in all tgds rather than adorn it with annotations to indicate any special join semantics.

2.2.3 Operations involving correspondences

What are meaningful operations for Youtopia users to perform on metadata relating to table-level and tuple-level correspondences? First, of course, users may create (and remove) tgds. However, we do not discuss such higher-order operations in this thesis; supporting them raises its own unique set of challenges and adds its own layer of complexity to the design, as even the following simple example shows:

Example 2.2.5. *Suppose we have two tables in the database with the same schema but completely different content, and tgds are created specifying that these two tables should be copies of each other. How should the system proceed? Should it copy all data from one table to the other and vice versa to create copies? Or should it enforce the tgds only for future inserts to the tables, in the sense that any insert to the first will be propagated to the second and vice versa?*

Therefore, we assume from now on that the relations in the database and the tgds themselves are fixed, and we focus on operations that are meaningful and necessary in that context. These operations come in two kinds: specifying (and removing) per-tuple connections, and enforcing tgds when violations occur due to other user operations on the database.

The first kind of operation is straightforward. Users should be able to specify that a pair of tuples is an explicit witness to the satisfaction of a tgd. They should also be able to undo this operation, i.e., bring the database back to a state where that particular witness is no longer an explicit witness. This is easy to do in a visual interface, where a user can draw a line between two tuples to specify that they correspond, or remove such a line if it is incorrect.

The second kind of operation has to do with tgd enforcement, and requires substantially more work to formulate fully. Why, to begin at the beginning, is tgd enforcement necessary? Data in the database will be modified during normal usage of the repository. Tuples will be inserted, deleted, modified, and – as we have seen – merged and/or split as part of duplicate-related data cleaning. (The latter two operations can be modeled as a set of inserts and deletes). Even if we begin with a database that satisfies all tgds, these operations will cause violations to occur.

Definition 2.2.6 (Tgd violation, join-free case)**.** *Let σ and the mappings μ be as in Definition 2.2.2. If σ is not satisfied, there is a set M of mappings μ that cannot be extended to corresponding μ' as explained in Definition 2.2.2. Every such μ is a* violation *of σ. Every corresponding $\mu(\mathtt{A}(\overline{x}, \overline{y})) \in D$ is a* violation witness *for σ.*

Thus, a tgd violation witness is a tuple that is present in the database and matches the LHS of the tgd, but does not come with a corresponding tuple to match the RHS. Again, the definition of violation and violation witness extend in a simple way to tgds with more than one atom per side. In this case, violation witnesses consist of a set of tuples from the database that *together* match the LHS of the tgd, but do not have a corresponding tuple or set of tuples to match the RHS.

Violations can be created by operations on the data in two different ways; information on how a violation was created can be important when deciding how and whether to correct it. Therefore, we distinguish between two kinds of violations.

Definition 2.2.7 (LHS-violations and RHS-violations, join-free case)**.** *Let σ and the mappings μ be as in Definition 2.2.6. Suppose σ is not satisfied on a database D, but was satisfied on the previous version of the database D', where D was produced from D' by a single insert, delete, modify, merge or split operation. Let μ be a concrete violation and $\mu(\mathtt{A}(\overline{x}, \overline{y})) \in D$ a concrete* violation witness *for σ. If $\mu(\mathtt{A}(\overline{x}, \overline{y})) \notin D'$, μ is a LHS-violation, otherwise it is a RHS-violation.*

Again, this definition extends to the multi-atom case. The intuition here is that LHS-violations are created by adding to the database a "new" tuple that matches the LHS of σ, but is (so far) missing a corresponding RHS tuple. For example, a tuple insertion can only ever cause a LHS-violation. On the other hand, RHS-violations are created by removing from the database a tuple which was previously part of a witness for the *satisfaction* of σ, so that the corresponding LHS tuple (or tuples, for a multi-atom tgd) from that satisfaction witness are "orphaned" and become a *violation* witness. A tuple deletion can only ever create RHS-violations.

How should LHS-violations and RHS-violations of tgds be handled? At a high level, for each violation type, there are four options:

- the violation can be ignored,
- the operation causing the violation can be disallowed (or reverted, if it has already happened),
- the violation witness can be removed from D,
- the violation witness and σ can be used to generate a tuple or set of tuples. These are inserted into the database and remove the violation by trans-

forming the violation witness into (part of) a satisfaction witness. This is essentially the standard chase procedure for tgds [14, 50].

The first two options are self-explanatory and simple to implement. The other two are less so, because they involve corrective operations that may themselves, in turn, cause violations that require correction. We discuss them in more detail shortly.

The choice of an enforcement mechanism for each violation type is ultimately a decision to be made by the system users. There are, however, certain choices that are better motivated than others. For example, correcting LHS-violations using the third method (removing the violation witness) is a questionable idea, as such a correction essentially reverts the user edit that created the violation witness in the first place. If this is the desired outcome, it would have been better achieved by disallowing the original edit. Similarly, correcting a RHS-violation using the fourth method would revert or almost revert the user operation that was the source of the violation, as in the following example.

Example 2.2.8. *Consider the database in Figure 1.1. Suppose the tuple* `R(Geneva Winery, XYZ Tours, Great!)` *is deleted. This creates a RHS-violation of* σ_3*. Correcting it using the fourth option above requires the insertion of tuple* `R(Geneva Winery, XYZ Tours,` x_3`)`*, which almost regenerates the tuple that was just deleted. If the user truly wants this tuple removed, they have no way to achieve this without figuring out by hand that they must first delete a tuple from either the* `A` *or* `T` *table.*

We now explain in more detail how the third and fourth enforcement method may be implemented; in view of the argument above, however, we restrict our discussion to applying the fourth option to LHS-violations and the third option to RHS-violations; in what follows, we refer to these processes as the forward and backward chase respectively.

The Youtopia forward chase

Superficially, the forward chase in Youtopia, as informally described in the above list, is very similar to the standard tgd chase [30]. A tgd violation witness is identified: matching RHS tuples are generated, and – usually – inserted into the database. The process then repeats. However, if the insertion just described were always carried out, it would sometimes be possible to generate an infinite cascade of inserts, as we have seen in Example 1.3.2. In the literature, this problem is usually handled by imposing a global acyclicity restriction on the tgds permitted in the system [47, 45, 39, 30]. Of course, this is not an appropriate solution in a community setting, as explained in Chapter 1.

However, tuple insertion is not the only way to repair a violation by supplying a matching RHS to the witness. Sometimes it is possible to provide a matching RHS by *unifying* some labeled nulls with other values in the database. Indeed, if a knowledgeable human were observing the infinite sequence of inserts in Example 1.3.2, they would very likely step in and short-circuit the process. For example, they might supply the additional information that the suggested airport for NYC is itself in NYC. This is equivalent to indicating that the two tuples $\texttt{C(}x_4\texttt{)}$ and `C(NYC)` are referring to the same fact and should be collapsed.

The Youtopia forward chase model is a formalization of the above intuition. A forward chase starts out in the traditional way: we identify violations and their witnesses, generate new tuples and insert them. This chase's execution sequence can be represented as a tree, with inserted tuples as nodes and direct causality relationships as edges. If on any path the system detects nondeterminism such as was present in our example with respect to the tuple `C(`x_4`)`, the chase stops along that path and awaits human intervention. Our notion of nondeterminism is based on the concept of the *more specific than* relation on tuples.

Definition 2.2.9 (Specificity Relation)**.** *A tuple* $t = (a_1, \cdots a_k)$ *is* more specific than *a tuple* $t' = (a'_1, \cdots a'_k)$ *if the map* f *defined as* $f(a'_i) = a_i$ *is a function and* f *is the identity on constants (i.e. values that are not labeled nulls)*

We say that nondeterminism occurs on a chase path if a tuple t belonging to relation R is generated by the chase, but R already contains a tuple t' which is more specific than t. In this case, it is possible that t is intended to represent the same fact as t', and t should be set aside for human inspection to determine whether this is the case. In the above example, the tuples `C(NYC)` and `S(`x_3`, `x_4`, NYC)` will be inserted by the chase, as no tuples more specific than these exist. On the other hand, the tuple `C(`x_4`)` will not be inserted, since more specific tuples do exist. In this way, the chase will stop even though the tgds are cyclic. Indeed, it is always the case that (this phase of) a Youtopia forward chase must stop sooner or later.

Lemma 2.2.10. *For any forward chase using the above algorithm, computation will stop along all paths in the chase tree after finitely many steps, unless the chase terminates before such a point is reached.*

This lemma is proved through a simple counting argument, making use of the fact that the starting database is finite and thus contains a finite set of variables and constants. The chase cannot generate tuples containing those forever; from some point onwards, it must generate only tuples containing fresh labeled nulls (i.e. those that were not in the initial database). After another finite set of steps, the tuples already generated will also cover/represent all possible ways in which labeled nulls can repeat within a tuple. Any tuple generated after this last point must unify with one already in the database.

Once a chase has stopped, it is time for a human user to step in and assist the violation correction process. The user has access to all the tuples which were generated but not inserted into the database; we refer to those as *(positive) frontier tuples*. Faced with a frontier tuple t, a user may perform one of two *frontier operations*:

- *expand* t, that is, insert t into the database.
- *unify* t, that is, choose another tuple t' in the same relation as t which is more specific than t and perform variable unification between any labeled nulls in t and t'. t disappears after such an operation.

Given a suitable interface that provides meaningful provenance information for the frontier tuples, these operations should be quite feasible for a knowledgeable human to perform. The user can simply be presented with a frontier tuple and asked: "Is this a new tuple, or can you match it to a tuple already in the relation?". If they answer yes to the first option, they are requesting expansion, and otherwise the matching tuple they indicate supplies the necessary unification information. In terms of the duplicate operations presented in Section 2.1, unification is a special instance of merge, and expansion of unmarking two tuples as duplicate candidates. Because of this, the frontier and duplicate management operations can be performed using the same interface.

The unification operation may cause changes to other tuples in the system if they contained one of the labeled nulls which disappeared in the unification. These changes may themselves cause further tgd violations. Expansion may also generate new violations due to the insertion of t. However, in both cases the new violations are guaranteed to be LHS-violations, so the chase can simply add them to its violation queue for future correction.

A special case for frontier operations concerns tgds with multiple atoms on the right-hand side. The firing of such a tgd in a forward chase generates multiple frontier tuples that may share some labeled nulls. On such tuple sets, the frontier operations work as expected given that the shared labeled nulls must be treated consistently.

After a frontier operation, the system may be able to carry out further deterministic chase steps; if it can, it does so until it terminates or once again reaches a point where it must stop on all paths. At this point it asks for user assistance again and the process repeats. A chase execution thus consists of a sequence of *deterministic strata* separated by periods of blocking and waiting for frontier operations. The full forward chase execution model is presented in Algorithm 1.

The Youtopia backward chase

The backward chase is a process that corrects RHS-violations by removing the violation witnesses from the database. Thus in Example 2.2.8, the violation of σ_3 would be corrected by deleting either `A(Geneva, Geneva Winery)` or `T(Geneva Winery, XYZ Tours)` – one is sufficient. The backward chase is therefore a process of cascading deletions.

Algorithm 1: The forward chase

```
writeSet := initial user operation
violQueue := $\emptyset$
repeat
  {Begin deterministic stratum}
  repeat
    perform writes in writeSet
    writeSet := $\emptyset$
    violQueue.remove(violations just corrected)
    violQueue.append(violations just created)
    choose unprocessed v $\in$ violQueue
    if v is deterministically repairable then
      writeSet := set of corrective writes for v
    else
      generate frontier tuples for v
      make nonblocking request to user for frontier op
    end if
    mark v as processed
  until violQueue.isEmpty $\vee$ all v await frontier ops
  {End deterministic stratum}
  if awaiting frontier ops then
    block while no frontier operations performed
    writeSet := result of first frontier operation received
  end if
until writeSet.isEmpty $\wedge$ violQueue.isEmpty
```

Unlike a forward chase, a backward chase must eventually terminate, as it cannot delete more tuples than exist in the database initially. However, backward chases come with their own flavor of nondeterminism which also requires human assistance to resolve. In our example, it is sufficient for one of the two tuples in question to be deleted for σ_3 to be satisfied again; the Youtopia system recognizes this, but does not make a decision, deferring it instead to a user.

Like the forward chase, the backward chase progresses deterministically as far as it can; when it encounters a situation like the above, with a set of tuples where at least one must be deleted, it marks all these tuples as *negative frontier tuples* and requests user assistance. Faced with such a set of negative frontier tuples, a user may perform the *negative frontier operation* of deleting any subset of the tuples.

Once again, performing this frontier operation requires no technical knowledge from the user. They are simply presented with a set of tuples and requested to select the subset which is to be deleted based on their domain knowledge.

We remark that the deletion frontier operation is in some sense the counterpart of the expansion operation on the positive frontier; both cause the frontier to advance further. It is also possible to formulate a negative frontier operation that would be the counterpart of unification; this would be a *reconfirmation* operation, where a user specifies for some proper subset of a set of negative frontier tuples that the subset is *not* to be deleted. Determining whether such an operation would be useful in a community DBMS and, if so, implementing support for it, is future work.

If the backward chase functionality is desired and implemented in a real-wold system, additional issues must be considered. Cascading deletions may be dangerous from an access control standpoint; the system must check whether a deletion may cascade into a table where it would cause a permissions violation. In such a case, it is preferable that the original deletion should be rejected by the system (following the second of our enforcement options). Moreover, the interface must make it easy for the user to determine whether they really intend to perform a deletion. In our ongoing discussion based on Example 2.2.8, the user's intent may have been just to remove the review rather than delete the entire tuple; if so, they should have replaced `Great!` by a fresh labeled null instead.

2.3 The Youtopia data cleaning operations

This section presents the concrete set of operations used in the Youtopia system; the set includes almost all the operations introduced in this chapter. As mentioned earlier, we only consider the case where all relation schemas and tgds are fixed; adding support for changing these is ongoing work. Algorithms 2 through 7 describe further bookkeeping actions that must be taken after each of our operations, as explained in the textual introduction which follows.

As discussed, we need to make design decisions on the enforcement of tgds. In Youtopia, LHS-violations are either corrected via the forward chase or ignored; the enforcement mechanism is a property of the specific tgd and is made explicit at the time of its creation. If the tgd has its LHS-violations corrected by the chase, we call it a *subscription*, otherwise we call it a *connection*.

To understand why we allow both connections and subscriptions in Youtopia, consider as an example the tgd σ_5 in Figure 2.2. If a user inserts a tuple relating

Algorithm 2: Actions to take after tuple insertion

```
{Tuple $t$ has been inserted into relation $R$}
**if** the insert occurred to satisfy a subscription $\sigma$ having $R$ on the RHS **then**
    {$t$ was inserted either by the chase or by a user performing the expansion operation using Algorithm 5}
    add explicit per-tuple correspondence information linking $t$ to the LHS tuples which "generated it" (i.e. belong to the same $\sigma$ satisfaction witness as $t$)
**end if**
**for all** connections $\sigma$ having $R$ on the LHS **do**
    prompt user to supply optional per-tuple connection information between $t$ and appropriate RHS tuples
**end for**
**for all** subscriptions $\sigma$ having $R$ on the LHS **do**
    V := violation witnesses for $\sigma$ containing $t$
    run the forward chase to correct all violations in V
    {any further inserts in the chase generate recursive calls to Algorithm 2}
**end for**
```

to a new biography into table `Biographies`, and the biography is of a person not in the `(Famous) People` table, what should the system do? Arguments can be made both for running the forward chase and for ignoring the violation. On the one hand, if the unknown person is the subject of a biography, they are probably important and should be in the `People` table, so a forward chase should be run. On the other hand, suppose the `People` table is a high-quality, master table which should not be corrupted with potentially spurious inserts, whereas the tuples in the `Biography` table generally come from questionable sources. Then it is better to

Algorithm 3: Actions to take after tuple deletion

```
{Tuple $t$ has been deleted from relation $R$}
for all connections or subscriptions $\sigma$ having $R$ on the RHS do
  for all former $\sigma$ satisfaction witnesses w that contained $t$ do
    let w' be the portion of w corresponding to the LHS of $\sigma$
    if w' is now a violation witness for $\sigma$ then
      notify user of this RHS-violation and allow them to correct it
      {any corrective actions generate calls to their own handling algorithms,
      e.g. a correction through a further delete requires a recursive call to
      Algorithm 3}
    else if w was an explicit satisfaction witness for $\sigma$ then
      update the per-tuple correspondence info for w to reflect the fact that
      while these tuples are still part of a $\sigma$ satisfaction witness, they are no
      longer part of an explicit satisfaction witness
      {For example, in Figure 2.3, if the second tuple in P is deleted, person_key
      must be set to 0 for the first tuple in B}
    end if
  end for
end for
```

leave the tgd partially unsatisfied than to have large numbers of inserts, potentially of low-quality data, be propagated into `People`. Indeed, σ_5 is very close to a traditional inclusion dependency, and those are never implemented with a forward-chase like enforcement mechanism, precisely because the RHS table in an inclusion dependency tends to be the "master" table in some sense. In fact, true inclusion dependency-style enforcement would disallow the new insert – a mechanism which

Algorithm 4: Actions to take after tuple modification

```
{Tuple $t \in R$ has just been changed to $t'$}
run Algorithm 3 as though $t$ had just been deleted
run Algorithm 2 as though $t'$ had just been inserted
**if** $t$ was a LHS tuple included in an explicit satisfaction witness w for some tgd
$\sigma$ **then**
  **if** the modification to $t'$ means $t'$ cannot be part of the same witness **then**
    {there was a change to attribute shared between a LHS and a RHS rela-
    tion, for example, suppose in Figure 2.3, the first tuple in B had its sub-
    ject_lastname changed to Washington }
    update the per-tuple correspondence info to reflect the fact that w is no
    longer an explicit satisfaction witness for $\sigma$
  **end if**
**end if**
```

Algorithm 5: Actions to take after duplicate unmarking

```
{Tuples $t$ and $t'$ have been unmarked as duplicate candidates}
**if** one of $t$ or $t'$, say $t$, was a frontier tuple **then**
  **if** no more tuples $t''$ are potential duplicates for $t$ **then**
    {perform frontier operation of expansion}
    insert $t$ into the database
    run Algorithm 2
  **end if**
**end if**
```

Algorithm 6: Actions to take after tuple merging

```
{Tuples $t$ and $t'$ have been manually merged into $t''$}
for all unifications between a labeled null $x_i$ and another value $v$ performed in the merge do
    O := set of tuple modifications required to replace all occurrences of $x_i$ in the database with $v$
    perform all operations in O, including subsequent calls to Algorithm 4
end for
for all w explicit satisfaction witnesses to $\sigma$ containing $t$ or $t'$ on the RHS do
    update the per-tuple correspondence info to reflect the fact that $t''$ is now the corresponding RHS tuple in all cases
end for
for all w explicit satisfaction witnesses to $\sigma$ containing $t$ or $t'$ on the LHS do
    update the per-tuple correspondence information for $t''$ accordingly, requiring the user to resolve any conflicts manually
    {a conflict can arise if there were two explicit satisfaction witnesses for $\sigma$, w containing $t$ on the LHS and w containing $t'$ on the RHS, but w and w' contain different RHS tuples; correspondences must be many-to-one}
end for
```

is almost certainly too strong for most collaborative DBMS settings. Ignoring the violation allows the inserts into `Biographies` to still proceed, but protects the data quality of `People`. Ultimately, only the user who understands the domain, the tables and the tgd can decide whether to make it a connection or a subscription.

RHS-violations, on the other hand, are not repaired automatically. The system does notify the user of them as they occur so that the user may *manually* cascade

Algorithm 7: Actions to take after tuple splitting

```
{Tuple $t$ has been manually split into $t'$ and $t''$}
for all w explicit satisfaction witnesses containing $t$ on the LHS or RHS do
  have the user update the per-tuple correspondence information manually
end for
for all violation witnesses of any subscription $\sigma$ containing $t'$ or $t''$ do
  have the user specify whether to correct the violation (via the forward chase)
end for
```

the deletion if they choose, but it takes no further action on its own. We believe that in the majority of cases, the backward chase would be too strong a mechanism. When users create tgds, they think of them primarily as assistance tools either for specifying tuple-level correspondences or for propagating inserts (in Youtopia, this functionality is provided by tgds that are correspondences and subscriptions, respectively), rather than as logical formulas that truly must hold on the entire database. In the majority of cases, a deletion that causes a RHS-violation of a tgd should create an explicit exception to the satisfaction of the tgd. If no exception is desired, it is reasonable to expect the user to correct the problem manually.

The Youtopia data cleaning operations are then as follows:

- *data manipulation*: **insert**, **delete** and **modify** tuples.
- *duplicate management*: **mark** and **unmark** any pair of tuples as duplicate candidates, **merge** two duplicate candidates into one tuple, **split** a tuple into two. This includes the forward chase frontier operations as a subcase.
- *correspondence management*: **specify** or **un-specify** a tuple-level correspondence between two (or more) tuples, with respect to a specific tgd.

In many cases, performing an individual operation from those listed above may affect metadata related to duplicates and per-tuple correspondence information; thus, additional actions may need to be taken by the system. Below, we present very high-level pseudocode explaining what is done in Youtopia for each operation that may require further action. Three of the operations do not require any further action - these are marking a pair of tuples as potential duplicates, marking two tuples as corresponding with respect to a tgd, and unmarking them as corresponding. Algorithms 2 through 7 present pseudocode for the remaining six.

As is clear from the pseudocode, the bulk of the work to be done is related to the maintenance of per-tuple correspondence information, which is frequently nontrivial. Indeed, in the case of the tuple split operation, we leave this work to be performed explicitly by the user, as there are arguably no corrective actions that are applicable in all possible situations. As a tuple split is a relatively complicated operation that requires substantial domain knowledge from the user, it is reasonable to expect them to be able to update the per-tuple correspondence information as well at the time of the split.

CHAPTER 3
TRANSACTION SERIALIZABILITY

3.1 Non-blocking algorithms for concurrency control

This chapter introduces a notion of serializability for Youtopia transactions; we call it Youtopia Multiversion Conflict-Serializability (YMCSR), as it is related to the classical Multiversion Conflict-Serializability (MCSR) [62]. The reader might be curious about the reasons behind our decision to work explicitly within the multiversion setting. After all, it explicitly includes implementation-related machinery (i.e. versioning), and is thus in a sense not as fully general as some other settings. We have chosen to work in the multiversion framework because is especially well-suited for the design of non-blocking concurrency control algorithms in unrestricted system scenarios (i.e. those which may include a high level of transaction contention with respect to data).

As mentioned in Section 1.3, any concurrency control algorithm that is practical for use in a community DBMS must be "non-blocking" in some sense. We begin by exploring and formalizing this concept of non-blocking. Most of the material in this section is closely related to facts known as folklore in the classical concurrency control community; our goal in this section is to formalize it and make precise the way in which it applies to Youtopia transactions, which have some non-standard features. Our formalization will serve not only to motivate our choice of the multiversion environment, but to set the stage for a more involved examination of the tradeoff between the strength of concurrency control enforcement mechanisms and the achievability of isolation levels in Chapter 4.

We start by formalizing the notion of a Youtopia transaction. As we have already mentioned, a transaction is sequence of reads and writes, optionally ended by an explicit special *commit*, *abort* or *terminate* operation. A transaction is not required to ever perform any of these; for example, a forward chase that is perpetually extended by frontier expansion operations might neither commit nor terminate. Such a chase might well arise in practice, for example in the genealogical database discussed in Example 1.3.3.

A word is in order on the explicit *abort* operation. It will occur when the system aborts a transaction because of concurrency control problems, but also whenever a user manually specifies that a concrete transaction is to be aborted and "undone" with respect to its effects on the database. A transaction may be aborted, for example, because is is no longer useful or relevant (e.g. a travel database user realizes that they cannot make a planned trip), or because a user is removing cases of vandalism on the data, or when a large portion of the transaction was carried out by an algorithm and a human subsequently determines that the algorithm used an inappropriate heuristic and made inappropriate changes to the data.

It is natural to specify the reads in a transaction intensionally, that is, by means of queries. Both the users and the algorithms running on the system (such as the forward and backward chase from Chapter 2) normally perform their reads through queries; thus, the specific tuples they read will depend on the state of the database at the time the query is posed. A priori we do not restrict the expressiveness of the query language; however, in any real-world system, we can expect most queries to be expressible in standard SQL. This is a realistic restriction to impose on the human users; also, as we will see in Section 3.4, it holds on all queries posed by the algorithms running the Youtopia forward and backward chase.

Writes, on the other hand, can be specified extensionally. In our model, a write is a tuple insertion, deletion or modification operation. We note that all the operations introduced in Section 2.3 can be modeled as a sequence of such basic writes. For insert, delete, modify, merge and split operations, this is self-explanatory. If we assume that per-tuple correspondence information is specified with extra attributes as in Figure 2.3, it is easy to see how the correspondence management operations can also be represented as tuple modifications. Finally, marking and unmarking pairs of tuples as duplicates can also be implemented with tuple insert and delete operations. For example, the **Dup** relation can be represented as an index table relating each tuple's key to the keys of its potential duplicates. In this representation, the translation from a duplicate mark/unmark operation to a set of tuple insertions and deletions on the index table is straightforward.

Definition 3.1.1 (Transaction)**.** *A* transaction *is a finite or infinite sequence of* operations *and* delays*. Each operation is either a* read query*, a tuple* insertion*,* deletion *or* modification*, or the special* commit*,* terminate *or* abort *operation. A transaction may contain only one* commit *or* terminate *operation; if it does contain such an operation, it must be the last operation in the transaction. The semantics of the read, insert, delete and modify operations are the standard ones. The* terminate *and* commit *operations have no direct effect on the database. A* delay *is a number representing time in a suitable unit and also has no effect on the database; delays may be arbitrarily large.*

We assume a system model in which transaction execution is controlled by a *scheduler* component that may permit an operation to run, or delay the operation execution until some future time.

Definition 3.1.2 (Transaction states)**.** *Consider a transaction T that is executing* *in the system. At a given point in time, T may be in one of three states. It is in the* running *state if a read or write operation is actively being carried out. It is in the* waiting *state if it is currently "executing" a delay, that is, it is waiting for user intervention to proceed. Otherwise, it is in the* ready *state – this means that it knows which concrete operation it wishes to perform next and is awaiting permission from the scheduler to execute it.*

Definition 3.1.3 (Non-blocking scheduler)**.** *We say that a scheduler is* non-blocking *if no transaction in the ready state is ever required to wait for an indefinitely long period of time before being allowed to run.*

The most important source of indefinite waits in Youtopia are, of course, the delays in the transactions themselves. Because these may occur at any time, a non-blocking scheduler may never delay the execution of a transaction T which is *ready* specifically in order to wait until another transaction T' exits its *waiting* state. We formalize this somewhat as follows.

Definition 3.1.4 (Blocking criterion)**.** *Consider a test ϕ; suppose ϕ is evaluated on a system where a set of transactions is in the* waiting *state and evaluates to* `false`*. Suppose now that the delay for each of the waiting transactions is artificially extended to an infinite time duration. If it is not guaranteed that ϕ must nevertheless evaluate to* `true` *after finitely many time units have passed, ϕ is a blocking criterion.*

The reader will note that our formalization is not complete: we use the term *test* without defining it. This is because we do not wish to restrict in any way the class of criteria or tests that a scheduler algorithm can evaluate on a running system. For example, a scheduler may wish to perform tests based on timing events ("has this transaction been in the *waiting* state for more than three seconds?"). A test can be considered as a subroutine that returns a binary value when run by the scheduler, and may read arbitrary information from the state of the running system; further formalization of this notion is not necessary for the present discussion.

The following lemma is obvious and connects Definitions 3.1.3 and 3.1.4.

Lemma 3.1.5. *A scheduler is non-blocking iff it uses no blocking criteria in determining whether or not to allow a transaction that is in the* ready *state to run.*

In Youtopia, the only acceptable schedulers are non-blocking, following Definition 3.1.3. Inspecting the definition clarifies that some amount of "blocking" in the system may in fact be acceptable – for example, transactions may be made to wait until a timeout occurs. However, no blocking may be performed based on any transaction's anticipated future re-entering of a *ready* state after it has been *waiting*. It should be clear that classical locking-based algorithms such as 2PL and its variants are unsuitable in Youtopia, as they use blocking criteria in deciding which transaction steps they allow to proceed. For example, some 2PL variants delay steps until a particular transaction terminates (and releases its locks).

How can concurrency control be enforced in a non-blocking way? Intuitively, it seems that the scheduler must allow transaction steps to proceed even when it cannot guarantee that it is "safe" for them to do so from a concurrency control standpoint; this, it appears, may be "risky" in some sense. We can formalize these intuitions as well.

A scheduler's purpose is ultimately to enforce some property on the transactions' execution. Normally this is a serializability property of some sort, and it is typically defined on *schedules*, which we now define for Youtopia transactions.

Definition 3.1.6 (Schedule)**.** *A transaction* schedule *is a pair (D, τ) where D is an initial database and τ is a finite or infinite sequence of operations that have been performed by the transaction on D. Read queries, tuple insertions and tuple deletions are included directly in the schedule; tuple modifications are represented as a tuple deletion followed by an insertion. When dealing with the operations within a schedule, we use the notation $w(t)$ for a write of tuple t, and $r(q)$ for a read that consists of posing the query q. A schedule can be finite or infinite. We denote by $\tau(D)$ the database that results when τ is actually executed over D.*

The above definition is for a single transaction. If multiple transactions execute and their operations interleave, the resulting sequence is a multi-transaction schedule. We can assume that in a multi-transaction schedule we can distinguish which operations were performed by which transaction, through suitable tagging of the operations. We will not differentiate between single- and multi-transaction schedules from now on.

Definition 3.1.7 (History)**.** *A* history *is a schedule in which every transaction performs (and ends with) either a* commit*, an* abort *or a* terminate *operation.*

We assume the scheduler is interested in enforcing some property ψ on the schedules in the system. Again, we leave the exact specification of this property open; normally, it will be strongly related to the desired notion of serializability. The only assumptions we make about ψ are the following:

- ψ holds on all empty schedules (i.e. schedules that contain no operations).
- its satisfaction is preserved under the prefix operator - if a schedule (D, τ) satisfies ψ, so does (D, τ') for all τ' which are prefixes of τ.

- it is nontrivial, that is, there exists at least one schedule (D, τ) such that ψ is false for (D, τ).

Definition 3.1.8 (Strict enforcement). *A scheduler* strictly enforces *a property of schedules ψ if at any point of the system's execution, the current schedule in the system is guaranteed to satisfy ψ.*

A scheduler that strictly enforces ψ is a good thing in general, as no transaction's operation ever needs to be reversed in order to guarantee that ψ still holds. Reversing an operation usually means a transaction abort, or at least a partial rollback. However, the following theorem tells us some compromises must be made when choosing a concurrency control algorithm, as non-blocking and strict enforcement are incompatible:

Theorem 3.1.9. *It is impossible for a non-blocking scheduler to strictly enforce a nontrivial property of schedules.*

Proof. The proof is by contradiction. Let (D, τ) be a schedule on which the property ψ is false – this must exist as ψ is nontrivial. Let τ' be the longest prefix of τ such that (D, τ') does satisfy ψ – in the worst case this is the empty schedule. Let the next operation that occurs in τ after τ' be o, performed by transaction number i. We can construct a running system scenario where all the transactions enter the *waiting* state after τ' is executed. Subsequently transaction i enters the ready state and requests that o be performed; all other transactions block indefinitely. If the scheduler allows o to proceed, it allows a violation of ψ, so it does not strictly enforce it. If it does not allow o to proceed, it will block indefinitely while transaction i is in the ready state, so it is not non-blocking. $\square$

We remark that Theorem 3.1.9 does not necessarily hold in a setting where the scheduler has additional information about the transactions, such as semantic knowledge that would allow it to determine what operations a transaction might still perform in the future (and more importantly, to know that certain operations will *never* be performed by a transaction in the future). Thus, in some restricted settings, it may be possible to have a non-blocking scheduler that does strictly enforce a property of schedules, but not in the general case.

In Youtopia, the non-blocking requirement cannot be dropped. This means that strict enforcement must be relaxed. Consequently, as the system runs, some schedules may be produced that do not satisfy the (still to be specified) property ψ. Because of this, a verification and correction mechanism is necessary to ensure that any violations of ψ are detected and repaired; as already mentioned, this normally involves aborting and restarting one or more transactions.

There are two broad possibilities for when the validation can occur. First, as in classical optimistic concurrency control, it is possible to wait until the transaction has terminated and then carry out validation for all the operations it has performed. The problem with using such a scheme in Youtopia is that transactions may never terminate. This means that for some transactions, a validation phase might never happen. If the system were using a classical optimistic concurrency control algorithm where each transaction maintains its private workspace until after the validation, this would also mean that a non-terminating transaction's writes would never become visible to anyone else. Consequently, such an enforcement mechanism is not suitable for Youtopia in general. There may, of course, be limited cases where it is adequate; investigating those is future work.

At the other extreme is the approach where validation is performed after every potentially "risky" operation, as in the classical *TO (timestamp ordering)* protocol. If two transactions are discovered to have interfered in an undesirable way, one is aborted. The number of aborts can be reduced substantially through the use of *versioning*: this mechanism hides certain transactions' writes from other transactions' reads and prevents a certain number of conflicts. This last option appears much more suitable for Youtopia, and it is therefore where we begin developing our concept of serializability. An interesting topic for future research would be to investigate a hybrid approach, where validation and correction are "somewhat" deferred – not to the end of a transaction, but to some suitable point determined either by system demands or the transaction semantics.

3.2 Youtopia Multiversion Conflict-Serializability

We now present our definition of Youtopia Multiversion Conflict-Serializability. It is related to classical MCSR, but with two important differences. First, our schedules specify the reads intensionally; this requires a new definitional framework. Second, we give a definition which assumes from the outset that a particular desired serialization ordering on the transactions is given: we define serializability of a schedule with respect to that ordering. In practice, when using our definition to design real-world algorithms, it is extremely likely that such an ordering would be natural and easy to obtain. For example, concurrency protocols such as MVTO assign transactions priorities based on the timestamp of the first operation and attempt to serialize with respect to those priorities. In the future, it may also be interesting to develop a broader definition of YMCSR which would not assume a particular serialization order.

A note is in order on the conventions associated with priority numbers. We assume, as is standard, that *higher*-priority transactions are those with *smaller* priority numbers. That is, the transactions should be serialized in *increasing* priority number order.

The first thing we need to do is to formalize versioning and version visibility. In our model, each operation on a tuple creates a new version, even if the same transaction has operated on the tuple already. (Deletions, of course, create special versions that indicate the tuple was deleted). Thus, it is natural to have two-component version numbers, the first being the transaction number and the second tracking the number of writes to this tuple within the particular transaction.

Definition 3.2.1 (Version number). *A version number $< i, j >$ is a pair of numbers i and j, which are either nonnegative integers or the special value ∞, which is needed for technical reasons. We also define an ordering relation $\prec$ on version numbers, as follows: given two version numbers $< i, j >$ and $< k, l >$, $< i, j > \prec < k, l >$ iff $(i < k \vee (i = k \wedge j < l))$. ∞ has the property that $i < \infty$ for every integer i.*

In a multiversion schedule, we can associate each write with the version of the tuple it produces; this is easily done by tagging the write with the version number, so that $w_i(t)$ becomes $w_i(t_{<i,j>})$. If a transaction performs an *abort* operation, all the versions it has created "disappear" from the database (either literally through being removed, or through some mechanism that makes them invisible to any read from any query).

The issue of reads is slightly more complicated, since they are specified intensionally; a read is not associated with a concrete version of a particular tuple. What we *can* say about a read query is that it is associated with a particular transaction, and this association determines the database which is *visible* to the query, as follows:

Definition 3.2.2 (Database visible to a query). *Consider a query q executed by transaction number i over database D. The part of D visible to q with respect to i is defined as follows. For every tuple t, include the highest-numbered version $< k, l >$ from D such that $< k, l > \prec < i, \infty >$.*

Definition 3.2.3 (Result of a versioned read query). *The result of a versioned read query $r(q)$ on a database D, when executed by transaction i, is the result of evaluating q over the part of D visible to q with respect to i.*

We can thus simply annotate reads with transaction numbers, as these fully determine which database is visible to the query.

Definition 3.2.4 (Multiversion schedule). *A multiversion schedule is a schedule in which each write of tuple t, $w(t)$, by transaction i is annotated as $w_i(t_{<i,j>})$ for some j, and each read $r(q)$ by transaction i is annotated as $r_i(q)$.*

We are now ready to define conflicts for operation pairs, as a prelude to defining YMCSR itself. In most versioned settings including ours, write-write operation pairs do not conflict, as there is never any overwriting of values and thus all writes commute with each other. This means all conflicts occur between pairs of operations involving one read and one write, as explained in Definition 3.2.5. The high-level intuition for the definition is as follows: consider the database at the time of the *second* of the two potentially conflicting operations. If on this database, the result of the read query is affected by the presence or absence of the result of the write, then the operations are in conflict.

Definition 3.2.5 (Conflicting pair of operations)**.** *Let (D, τ) be a multiversion schedule that includes operations $w_i(t_{<i,j>})$ and $r_k(q)$ with $i < k$. That is, τ has the form $\tau' \cdot o_1 \cdot \tau'' \cdot o_2 \cdot \tau_3$, where o_1 and o_2 are the read and write operation (not necessarily respectively). Define databases D' and D'' as explained below. o_1 and o_2* conflict *iff $q(D') \neq q(D'')$.*

If o_1 is the write $w_i(t_{<i,j>})$, let $D' = [\tau' \cdot o_1 \cdot \tau''](D)$, that is, the database that exists just before the read o_2 is performed. Let D'' be identical to D', except that it does not include the tuple version $t_{<i,j>}$ (i.e. we are making the outcome of the write invisible).

If o_1 is the read $r_k(q)$, let D' be $[\tau' \cdot o_1 \cdot \tau''](D)$ excluding *the results of any writes (i.e. any tuple version) produced by transaction k* after *$r_k(q)$ was performed. Let $D'' = o_2(D')$.*

This definition is very clear in the first subcase, where the read is the second of two operations: there is a conflict iff the result of the read would have been different had the write not happened. Indeed, we can use this subcase to define a reads-from (RF) relation on transactions in a schedule that keeps track explicitly of *read dependencies* between transactions:

Definition 3.2.6 (Reads-from relation)**.** *Let (D, τ) be a multiversion schedule including transactions i and j, with $i > j$. We define a reads-from relation, RF, on transactions, so that $RF(i, j)$ iff the schedule includes a write by transaction j followed by a conflicting read by transaction i.*

The second subcase in Definition 3.2.5 applies to those situations where the write comes second:

Definition 3.2.7 (Retroactively affecting a read). *If a read and write operation are in conflict within a schedule and the write occurs after the read, we say that the write retroactively affects the read.*

The reader might wonder about Definition 3.2.5 here. Clearly the conflict check must be performed, but why should it be performed on the particular databases we define? Understanding this case is easier if we keep in mind that our ultimate purpose is to maintain the illusion of serial execution in priority order. That is, the write in question is "supposed" to happen before the read. If the read happens earlier, it is in some sense "premature"; it was performed on a database which was "incomplete" with respect to writes by higher-priority transactions. This may cause a problem, and it turns out that the test in Definition 3.2.5 is guaranteed to detect a problem if it occurs (the converse is not necessarily true, as we will explain later).

Consider the read $r_k(q)$, and consider τ' in Definition 3.2.5. This is the part of the schedule containing the operations carried out on D before $r_k(q)$ happened. Some of these writes may belong to transactions numbered $l > k$; their results will not be visible to $r_k(q)$, so we can exclude them from consideration. Let τ'_k be the schedule obtained from τ' by removing all operations belonging to such lower-priority transactions. When $r_k(q)$ is posed on $\tau'(D)$, the visible database for the query is the same as it would be on $\tau'_k(D)$.

Once $r_k(q)$ has been performed on $\tau'(D)$, the die is cast. We need to ensure that this query result is the same as it would have been if $r_k(q)$ had been performed after all transactions numbered less than k had terminated. That is, suppose we knew that all transactions numbered $m < k$ would eventually terminate, and moreover we knew the future execution schedule for all these transactions; call this $\tau^t_{<k}$.

Then we need to ensure that $q(\tau'_K(D)) = q([\tau'_k \cdot \tau^t_{<k}](D))$.

Of course, we do not know $\tau^t_{<k}$, nor are we ever guaranteed to know it if the transactions do not terminate. However, if $q(\tau'_k(D)) \neq q([\tau'_k \cdot \tau^t_{<k}](D))$, then there is a finite prefix $\tau^f_{<k}$ of $\tau^t_{<k}$ such that we also have $q(\tau'_k(D)) \neq q([\tau'_k \cdot \tau^f_{<k}](D))$. That is, the change in the result of the read query will become apparent after finitely many future operations by higher-priority transactions have occurred.

This is exactly what the conflict test in Definition 3.2.5 is checking for; it is testing whether $q(\tau'_k(D)) = q([\tau'_k \cdot \tau^f_{<k}](D)$, where in our case $\tau^f_{<k}$ includes all the writes seen since $r_k(q)$ from transactions with numbers $< k$. The actual database D' in the second subcase of Definition 3.2.5 is described somewhat differently from $[\tau'_k \cdot \tau^f_{<k}](D)$ – we allow D' to contain versions created by transactions with numbers greater than k. However, no such versions will be visible to $r_k(q)$ anyway, so the two databases are equivalent for the purpose of the test.

We are now ready to define serializability. Classical serializability definitions frequently apply only to histories, rather than schedules in general; however, the property in the definition below can apply equally well to schedules that are not histories. As we are very likely to deal with the latter kind of schedules in Youtopia, it makes sense to extend the definition to include them as well.

Definition 3.2.8 (Youtopia Multiversion Conflict-Serializability). *A multiversion schedule is Youtopia Multiversion Conflict-Serializable iff it includes no writes that retroactively affect some read, and no reads that are affected by writes of aborted transactions.*

We briefly explain the connection of this definition to classical (multiversion) conflict serializability. Conflict serializability normally restricts the ordering of conflicting pairs of operations to be the same as that in some serial schedule. In our case, the serialization ordering is known up front; in our desired serial schedule, all conflicting read-write pairs are ordered so that the write occurs before the read (recall that to conflict with a read by transaction i, the write must be performed by a transaction with number $j < i$). Thus any instance of a write retroactively affecting a read is a pathological phenomenon and a serializability violation, as it is a pair of conflicting operations that are not ordered in the same way as they would appear in the serial schedule.

We now relate YMCSR to another classical notion – final-state serializability. As the latter is normally defined on histories only, the result that follows is given for Youtopia histories rather than general schedules. We also assume that no transactions abort in the schedule – this is again standard in serializability theory, where issues of recovery are typically separated from the properties which one tries to enforce in a "normal" (abort-free) setting. However, this same theorem could be proved for schedules that do involve aborts (recall that a "cleanup" mechanism is in place to hide all versions produced by aborted transactions from future reads as soon as the abort occurs).

Theorem 3.2.9. *Let (D, τ) be a Youtopia multiversion history in YMCSR, and assume it contains no transactions that perform the* abort *operation. Then (D, τ) is final-state serializable. That is, let τ' be the history obtained from τ by reordering operations so that all the operations belonging to the highest-priority (lowest-numbered) transaction come first, the second-highest-priority transaction next, and so on (assume that the reordering preserves operation ordering within a transaction). Then running either schedule on D produces the same final database: $\tau(D) = \tau'(D)$.*

Proof. We make the standard assumption that transactions are deterministic, in the sense that if a transaction runs twice and obtains the same results to its read queries on both runs, it will make the same writes.

We consider small rearrangements of the operations in τ. Specifically, let o be the first operation in τ which is "out of order" with respect to the serial ordering of transactions. Consider the history τ^* obtained from τ by moving o forward in the schedule into its "proper place". We show that $\tau(D) = \tau^*(D)$ and that (D, τ^*) is in YMCSR. As τ can be transformed into τ' by a finite sequence of such operations, the theorem will follow.

The simpler of the two cases occurs when o is a read operation, $r_k(q)$. Moving the read forward will not change its answer – we are only moving it past writes of higher-numbered transactions that were not visible to the read in the original schedule anyway. Thus transaction k will perform the same writes as it did in τ.

We argue that no other transactions' reads (or writes) will be changed either. Transactions numbered $< k$ are easy to deal with as, their reads and writes are all already ordered before o anyway (by our assumption that o was the first out-of-order operation). As for transactions numbered $> k$, this is seen through an inductive argument. Let i be the smallest transaction number greater than k. Transaction i reads from transactions numbered k or less; we have seen that none of these transactions' writes change, so nor do i's reads, or, by our determinism assumption, i's writes. So, by induction, all transactions still perform the same writes and $\tau(D) = \tau^*(D)$.

As for (D, τ^*) belonging to MCSR, the only way this could fail would be if some read were now retroactively violated by a write. The only read that has changed its position in relation to any writes is o, and – as we have already mentioned – the change in position was only with respect to writes by higher-numbered transactions anyway (those can never be in conflict with o). Thus o is fine, and as no other reads or writes have been moved, we are done.

Now suppose o is a write by transaction k. Again, it is clear that the writes of transactions numbered $<= k$ do not change. Let l be the smallest transaction number greater than k, and suppose the rearrangement causes the result of some $r_l(q)$ to change. This must mean that o used to occur after the read but has been moved before it. However, all operations of transaction k which are subsequent to o must still follow $r_l(q)$ in the schedule, since we have moved just one operation. From this, it follows that in the original schedule τ, o retroactively violated the result of $r_l(q)$ and (D, τ) was not in YMCSR - a contradiction. So, again generalizing this to an inductive argument, no other transaction's reads (or writes) change, and again $\tau(D) = \tau^*(D)$.

Finally, we argue that (D, τ^*) is still in MCSR. If not, this means some read $r_l(q)$ is now retroactively affected by a write $w_m(t_{<m,n>})$. In the original τ, where could o have been in relation to $r_l(q)$ and $w_m(t_{<m,n>})$? It cannot have occurred before the read, otherwise the retroactive violation would also be present in τ. So o must have occurred between the two operations or after $w_m(t_{<m,n>})$.

Suppose it occurred between the two operations. Denote the database on which $r_l(q)$ was originally performed (in τ) by D^*. A further set of writes by transactions numbered $< l$ may have occurred between the read and o; call that set W_1. Also, a set of writes may have occurred between o and $w_m(t_{<m,n>})$; call that set W_2. We know that neither o nor $w_m(t_{<m,n>})$ retroactively affected $r_l(q)$ in τ; this tells us that the answer to $r_l(q)$ is the same whether it is posed on any of the following databases: D^*, $W_1(D^*)$, $[W_1 \cup \{o\}](D^*)$, $[W_1 \cup \{o\} \cup W_2](D^*)$, $[W_1 \cup \{o\} \cup W_2 \cup \{w_m(t_{<m,n>})\}](D^*)$. Now, in τ', at the time $w_m(t_{<m,n>})$ is performed, the database visible to $r_l(q)$ is still $[W_1 \cup \{o\} \cup W_2](D^*)$, even though the write $\{o\}$ was performed before W_1 this time. Thus we know the result of $r_l(q)$ cannot be changed by the write $w_m(t_{<m,n>})$. The key observation we are using here is that in a versioned setting, writes are completely commutative with respect to their effect on the database.

The argument for the final subcase – o occurs after $w_m(t_{<m,n>})$ – is very similar. If $w_m(t_{<m,n>})$ were now retroactively violating the result of $r_l(q)$ in τ^*, then o would have retroactively violated the result of $r_l(q)$ in τ. So we are fine in this case also, and consequently τ^* is in YMCSR as required. □

We note that the converse of Theorem 3.2.9 does not hold; as mentioned before, our conflict-serializability criterion is a conservative one.

Example 3.2.10. *Consider the history (D, τ) where D is the empty database and τ is*

$$r_2(\texttt{SELECT * FROM R})\ w_1(t_{<1,1>})\ w_1(t_{<1,2>})\ terminate_1\ terminate_2$$

where tuple t is `R(a,b,c)`, and the first write inserts it into the database, while the second write deletes it. The first write retroactively affects the read by transaction 2, so the history is not in YMCSR. However, $\tau(D)$ is the same as the final database produced in a serial execution, so the history is final-state serializable.

3.3 Enforcing YMCSR

In this section, we begin the discussion of how YMCSR may be enforced in practice. When presenting our algorithms, we make the assumption that no transactions in the system perform the *commit* operation. This means we only need to enforce YMCSR itself, rather than also having to ensure the durability of those transactions which have committed. Indeed, a straightforward corollary of Theorem 3.1.9 is that a nonblocking scheduler *cannot* enforce both durability and YMCSR, as commits cannot be delayed. In the remainder of this chapter, then, any transaction, even one that has executed a *terminate* operation, may be aborted and redone by the system if necessary to enforce YMCSR. We will return to this "no durability" assumption and discuss it at greater length in Chapter 4.

The obvious non-blocking way to enforce YMCSR is to use an adaptation of the classical MVTO (multiversion timestamp ordering) algorithm [62]. The most general adaptation – in the sense that it makes as few design decisions as possible – yields the algorithm template given in Algorithm 8. The algorithm makes use of the RF^* relation, which is the transitive closure of the RF relation introduced in Definition 3.2.6.

The remainder of this section explains the design decisions that must be made to instantiate Algorithm 8 into a real, full algorithm for concurrency control.

First, we need to decide how transaction priorities are assigned. A common practical method is to assign them based on the timestamp of the arrival into the system of the first operation of the transaction, but this is by no means the only option; some transactions may inherently be worthy of a higher priority, because of factors such as the rank of the user starting the transaction or because the

Algorithm 8: Non-blocking scheduler template for enforcing YMCSR

```
choose next transaction operation $o$ to schedule
if $o$ is a read query $q$ then
  execute $q$, store it for future checks
  update the $RF^*$ relation appropriately
else
  {suppose $o$ belongs to transaction with priority $j$}
  execute $o$
  for all stored read queries $q$ of transactions numbered $i > j$ do
    if $o$ retroactively changes the result of $q$ then
      abort transaction $k$, where $k$ is either $i$ or $j$
      abort any transactions numbered $l$ such that $RF^*(l, k)$
      restart all just-aborted transactions with new priority numbers, larger
      than any already in the system
    end if
  end for
end if
```

transaction performs a particularly “important” task in the system.

Second, in the case of a conflict between a reader and a writer, the above template does not specify which transaction must be aborted. In standard MVTO, writers are aborted, on the assumption that read transactions are likely to be much more frequent and shorter, and should therefore be privileged for greater system throughput. On the other hand, an argument can be made that it is the reader who should be aborted, since – after all – it has lower priority. In practice, this is a decision that must be made based on the specific system deployment and workload.

Indeed, there is no reason for the choice to be made consistently each time; the algorithm can run a complex choice procedure to determine which transaction to abort. For example, it might choose the transaction whose abort would generate the fewest cascading aborts.

The third design decision concerns the implementation of the test for checking whether a read and a write conflict, as described in Definition 3.2.5. This test is necessary when checking whether a write retroactively violates a read, and when updating the relation RF^* after a read. However, it is not in general easy to carry out the test both precisely and quickly. If the read queries involved are complex enough, evaluating it can actually require at least one query evaluation on the database. Thus, it may be better in some cases to fall back on a coarse, conservative approximation rather than precisely performing the conflict tests, trading accuracy and an increase in the number of cascading aborts for time.

Finally, the first line of Algorithm 8 involves a scheduling choice among the transactions which are currently in the *ready* state. Ideally, of course, the scheduler should allow operations to run as soon as possible, while requiring as few aborts as possible. This means that the number of conflicts must be reduced as far as possible. If semantic knowledge is available about the transaction programs and the algorithm has some additional information about the possible future reads and writes, this may be helpful in scheduling. In the general case, where no such knowledge is available, it may be possible for the scheduler to at least develop heuristics about expected operations, using information about previous transaction behavior in the system. Based on these heuristics, transactions less likely to conflict can be allowed to interleave more aggressively or even run in parallel.

Also, when a transaction T enters the *waiting* state, the scheduler will normally permit other transactions' operations to run without waiting for T to become *ready* again, as explained earlier. However, as we have mentioned, sometimes the scheduler may choose to "wait for T a little" i.e. delay execution of other transactions for a bounded period of time. Such a decision will depend on the expected costs and benefits; for example, if the transaction is expected to have short delays, the scheduler may well choose to wait. This is particularly true if T is "worth waiting for" – because, for example, many running transactions with lower priority than T are likely to read from the relations that T is expected to write to.

3.4 A case study

This section is devoted to an extended case study in which we present a concrete instantiation of Algorithm 8. For simplicity, our case study takes place in a system where only a subset of the user operations from Chapter 2 are allowed. These operations are tuple insertion, deletion, and a limited form of tuple modification which we refer to as *null-replacement*. This is the replacement of all occurrences of a labeled null in the database by another value – either another labeled null or a constant. Null-replacements are interesting in that they are a kind of tuple modification that can only create LHS-violations of tgds, not RHS-violations. In the system we consider, LHS-violations are repaired by the forward chase and RHS-violations are repaired by a backward chase. This is an interesting scenario as it involves potentially long chains of both insertion and deletion operations propagating in the system simultaneously and interleaving, thus leading to a potentially large amount of interference.

3.4.1 Modeling the chase

First, we explain how precisely an execution of a Youtopia forward or backward chase maps onto the transaction model given in Definition 3.1.1. Since the chase may include long stretches of reads and writes that are generated automatically, it is important to understand what those reads and writes are going to look like; such knowledge is instructive in itself, but it also allows us to gear our implementation to the specific semantics and execution features of the chase. Issues like the computation of the RF^* relation are much easier in a setting where the kinds of read queries that may occur are quite restricted and known in advance.

Chase steps

First, we need to model the chase process in a way that exposes the read and write operations. To this end, we move from the model in Algorithm 1 to a model where a chase is a sequence of *steps*. Each step may or may not include a delaying (blocking) operation where input is requested from a user. Algorithm 9 gives our model for the execution of a single chase step. Here and in the remainder of this section, the term *chase* refers to any Youtopia chase, forward or backward, unless explicitly specified otherwise.

Algorithm 9 exposes the write and read operations a chase step performs as it is executed. The write operations occur first, at the beginning of every step; each write operation is either a tuple insertion, a tuple deletion, or a tuple modification which is part of a null-replacement. Subsequently, the transaction step may perform reads for two reasons: to determine what new violations were caused by the writes, and to correct the violation `nextViol`. This is explained in detail in

Algorithm 9: A chase step in Youtopia

```
perform a set $W$ of write operations
for all tgds $\sigma$ potentially violated by a write in $W$ do
  violQueue.append(new violations of $\sigma$)
end for
if violQueue contains a deterministically repairable $v$ then
  {step belongs to deterministic stratum}
  nextViol := $v$
else
  choose nextViol from violQueue
end if
generate a set $W'$ of corrective writes to repair nextViol
{generation may include blocking request for user input}
for all $v' \in$ violQueue which will be repaired by $W'$ do
  violQueue.remove($v'$)
end for
```

Section 3.4.1; for now, note as an example that correcting a LHS-violation requires reading the database to determine whether it contains any tuples more specific than the frontier tuple.

The presentation of Algorithm 9 reflects a simplifying assumption we make about the execution of a Youtopia chase – we assume all the writes are performed before the reads begin. This is reasonable: in the chase, the data read in a given step is virtually certain to have been modified by the writes that were performed at the start of the same step. Delaying the reads until the writes have completed is therefore necessary for correctness.

```
SELECT * FROM (LHS query)
WHERE NOT EXISTS (SELECT * FROM (RHS query))
```

Figure 3.1: Violation query template

```
SELECT * FROM A, T
WHERE A.name = T.attraction AND
AND A.name = 'Geneva Winery' AND T.company = 'XYZ'
AND NOT EXISTS (SELECT * FROM R
                WHERE R.company = T.company
                AND R.attraction = A.name)
```

Figure 3.2: Violation query example

Read queries

When a transaction executes a chase step, it poses read queries for two different reasons: to identify new tgd violations caused by a write, and to obtain information required for tgd violation correction. In the former case, the query to be asked for each tgd σ has the form presented in Figure 3.1.

`LHS query` and `RHS query` are conjunctive queries whose structure is dictated by σ and bindings by the newly written tuple. We refer to this type of query as a *violation query.*

Example 3.4.1. *Returning to our database in Figure 1.1, if the tuple `R(XYZ, Geneva Winery, Great!)` is deleted, the query to discover violations of σ_3 is shown in Figure 3.2. This query returns all the pairs of `A` and `T` tuples which have been affected by the deletion with respect to satisfying σ_3.*

A transaction may also perform queries in order to determine how to correct a violation, with or without human help. In the case of RHS-violations, no further reads are performed - the system or a human chooses a tuple to delete from one of the already-read violation witness tuples. For LHS-violations, however, additional reads may be performed because of the possibility of correction through unification. Given a set of frontier tuples for a violation, the system must perform the following queries for each frontier tuple t belonging to relation R:

- find any $t' \in R$ more specific than t
- if such t' are found, for all labeled nulls x in t which were *not* freshly generated when t was created, find all tuples in the database containing x. If the frontier unification operation is performed on t, these tuples must be modified.

We call these types of queries *correction queries.*

3.4.2 The algorithm

We now explain how we resolve the design decisions indicated in Section 3.3. First, in terms of transaction priorities, we use a timestamp ordering assignment, giving transactions priority numbers in the order of their arrival into the system. Second, when handling conflicts between a reader and a writer, we always choose to abort the reader, because it is the lower-priority transaction. The way we resolve each of the other two decisions requires more detailed discussion.

Scheduling

We first partially instantiate the scheduling policy in Algorithm 8 to place an interleaving restriction on steps from different transactions. We assume that interleavings are only permitted at the chase step granularity. That is, if a transaction has started a chase step and is performing writes, the scheduler will allow it to finish the writes and perform all the reads it requires in that step before any other transaction may proceed with any operation.

In practice, the scheduler's queue of transactions that are ready to take a step is populated asynchronously, as transactions complete frontier operations and return control to the scheduler. This means that chase steps may indeed be scheduled while another transaction's step is waiting for frontier operations. However, maintaining the illusion that interleaving only occurs at chase step level *as far as reads and writes are concerned* is not difficult. A chase step performs all its writes and violation read queries before asking for user input and blocking, so those cannot pose a problem. A step of a backward chase performs no further reads. A naïve implementation of the forward chase step may perform some correction queries after the user has performed a frontier operation. However, these can only be correction queries that ask for all tuples containing a given variable, and they are known to the system before the frontier operation is requested. Therefore, they can be performed beforehand as well. With a suitable index mapping variable names to tuples, the queries should not be expensive to perform, and the only remaining thing the system needs to ensure is that such a query is *not* logged if the user does *not* choose unification.

At a higher level, the scheduler has many options in terms of how it allows the chase steps to interleave. It may, for example, permit chases to run an entire deterministic stratum before scheduling a step from another transaction. As deterministic strata involve no delays, this is perfectly acceptable in a non-blocking scheduler. Again, in a real-world system, the choice of a scheduling policy would be made based on known performance and load data. In our implementation and experiments, we have chosen a basic scheduling policy that is one of the simplest to understand and implement. Our scheduler allows arbitrary step interleaving rather than running entire deterministic strata. It uses a round-robin decision procedure to determine which *ready* transaction's step is run next; just one step is run, and then the scheduler moves on to the next *ready* transaction.

Determining conflicts

As mentioned before, determining whether operation pairs conflict according to Definition 3.2.5 and computing the RF^* relation can be nontrivial for queries that are complex enough.

Correction queries are easy to handle in conflict tests without accessing the database. This is because a given tuple write affects the answer to a correction query either on all databases, or on none. For example, if a correction query asks for all tuples containing variable x_2, a write affects the answer iff the tuple written contains x_2.

Violation queries are more challenging; for them, the conflict test does require accessing the database to evaluate at least one query. Fortunately, however, a write can only change the answer to a violation query in a limited number of ways. For example, an insert can do so in two ways. It can contribute to the creation of a join result among relations on the LHS of a tgd, so that a new violation witness appears. Alternatively, it can provide the last tuple that makes a tuple appear in the join of relations on the RHS of some tgd. If the new RHS join tuple matches a previously existing violation witness, a violation is removed. Based on the type of the write (insert or delete) it is therefore possible to perform the conflict test by posing a single query which combines the original violation query with information about the new tuple. We treat tuple modifications (conservatively) as a delete followed by an insert.

In this case study, when we test for conflicts to determine whether a write retroactively affects a read, we always perform the precise conflict test. However, when computing RF^* for the purpose of determining cascading aborts, we consider three different algorithms.

When a transaction with priority number i aborts, the first naïve, strawman algorithm (`NAÏVE`) also aborts all transactions numbered $j > i$. This is sufficient to guarantee correctness, but is clearly not optimal.

Computing RF^* more carefully can be done in at least two different ways, one more precise and more expensive than the other. As mentioned, correction queries are the easy case: given a logged list of all previous writes, it is easy to determine which ones affect the result of a given read. If the list is kept in memory, the dependencies can be computed without querying the database. Violation queries, however, cannot be processed so simply without sacrificing precision.

The simpler of our two algorithms, `COARSE`, does not query the database to identify conflicts caused by violation queries. Whenever it processes a violation query that involves a set of relations $\{R_1, R_2, \cdots R_k\}$, it assumes that any transaction which has previously written any tuple to one of the R_is may be the source of a conflict, and includes this transaction in RF^*. This is a conservative overestimate of the real RF^*, so correctness is guaranteed.

The second algorithm, `PRECISE`, trades off run time for precision. `PRECISE` determines accurately, for each violation query q, which previous writes have changed the answer to the query, using the precise conflict test described above.

`COARSE` has linear time complexity in the number of writes performed so far on the database by updates which may still be aborted. For `PRECISE`, the dominating contribution to the complexity is the cost of the joins in the queries; these joins are dictated by the tgds (as in Example 3.4.1). In the worst case, their time complexity is polynomial in the size of the database and exponential in the join arity, i.e. in the number of atoms per side of a tgd. However, since the join predicates are a function of the tgds and thus known up front, it is possible to improve performance by appropriate indexing and join implementation.

3.4.3 Experiments

We have implemented Algorithm 8, instantiated as described above, and used each of `NAÏVE`, `COARSE` and `PRECISE` for computing RF^* and for determining cascading aborts. We now present a set of experiments that compare the performance of these three algorithms with respect to the number of cascading aborts and execution time.

Since Youtopia's paradigm of community database management is new and there are no real-world datasets for us to benchmark our algorithms against, we have used synthetic data and tgds, as explained below. Further, we needed to find a way to simulate frontier operations. Our code does this by choosing an option uniformly at random among all available alternatives for a frontier operation. In practice, this has the additional advantage of making all chases terminate, even when the set of tgds is cyclic: a unification (rather than expansion) operation is chosen sooner or later on every forward chase path.

Our experiments are run on a database of 100 relations, each randomly generated to have between one and six attributes. The relations are connected by tgds; each tgd is created by choosing a random subset of one to three relations for the LHS and another for the RHS. Smaller sets have higher probability, as humans are highly unlikely to create tgds with more than one or two atoms on either side. The remaining step in tgd generation is the choice of variables in the atoms; this is done randomly, with care taken to ensure that the tgds contain inter-atom joins as well as constants. Any constants used come from a small (size 50) fixed set of random strings.

Generating the initial database is performed using our own forward chase, with simulated user interaction; it is not easy to obtain an interesting database that satisfies an arbitrary, potentially cyclic, set of tgds using another method. We generate ten thousand initial tuples. The relations receiving those tuples are chosen uniformly at random, and the attribute values come from the same set of constants that was used in tgd generation. By keeping the domain of attribute values small, we ensure that joins between relations are highly likely to be nonempty, and that tgds are consequently highly likely to fire if these tuples are inserted. We insert these initial tuples into the database; each insertion sets off a forward chase which only ends when all constraints are satisfied.

We test our algorithms in several settings which vary in the number of tgds. We vary this number from 20 (a sparse setting) to 100 (a dense one). Settings with denser tgds are likely to exhibit longer chase runs, more writes, and therefore more conflicts and aborts; indeed, this is borne out by our results. In our runs, the set of tgds we used is monotonically increasing – that is, our experiments with 40 tgds involve the tgds used for the run with 20 tgds as well as 20 others, and so on. In all cases, the initial database is the same and satisfies all 100 tgds.

We show results on two workloads, each of 500 transactions. The first consists entirely of (chases induced by) inserts, the second of eighty percent inserts and twenty percent deletes. Each transaction in each workload is started by an insert or delete operation generated randomly and independently. First, the receiving relation is chosen uniformly at random. In the case of inserts, the values in the inserted tuples are chosen with equal probability to be fresh or from the previously mentioned set of constants. In the case of deletes, the tuple to delete is chosen uniformly at random from the relation. In the mixed insert/delete workload, the order of the transactions is then randomized to ensure that the runs do not involve alternating large batches of inserts and deletes. As already mentioned, the scheduling algorithm used in all our experiments uses a round-robin policy that interleaves chases at the level of individual steps. All runs are allowed to run to termination, and each data point is obtained as the average of 100 runs.

Our results are shown in Figures 3.3 and 3.4. The first graph of each figure shows the total number of aborts encountered during the run. Clearly, both `COARSE` and `PRECISE` outperform `NAÏVE` significantly. We only show the first few points for `NAÏVE`, as the huge performance difference is apparent even with very sparse tgd sets.

The second graph shows the total number of *cascading abort requests* during each run. This is the number of times during the run that the algorithm requests an abort even though the transaction involved is not in direct conflict with a just-performed write. Thus, this is the number of *purely cascading* aborts requested. It does not have a direct correspondence to the total number of aborts observed during the run. This is because aborts are not performed as soon as they are made necessary by a write, but only once control is returned to the scheduler. In

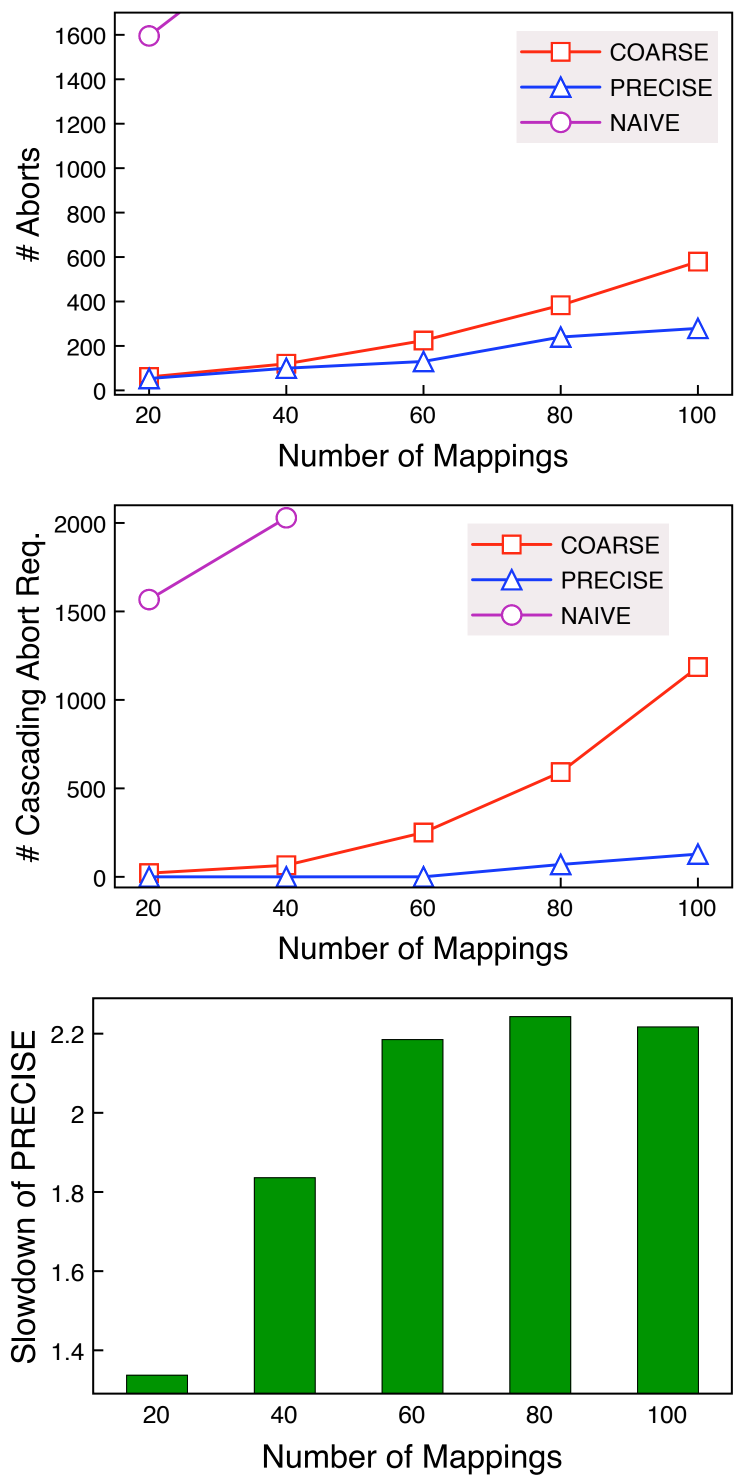

Figure 3.3: Results for the all-insert workload

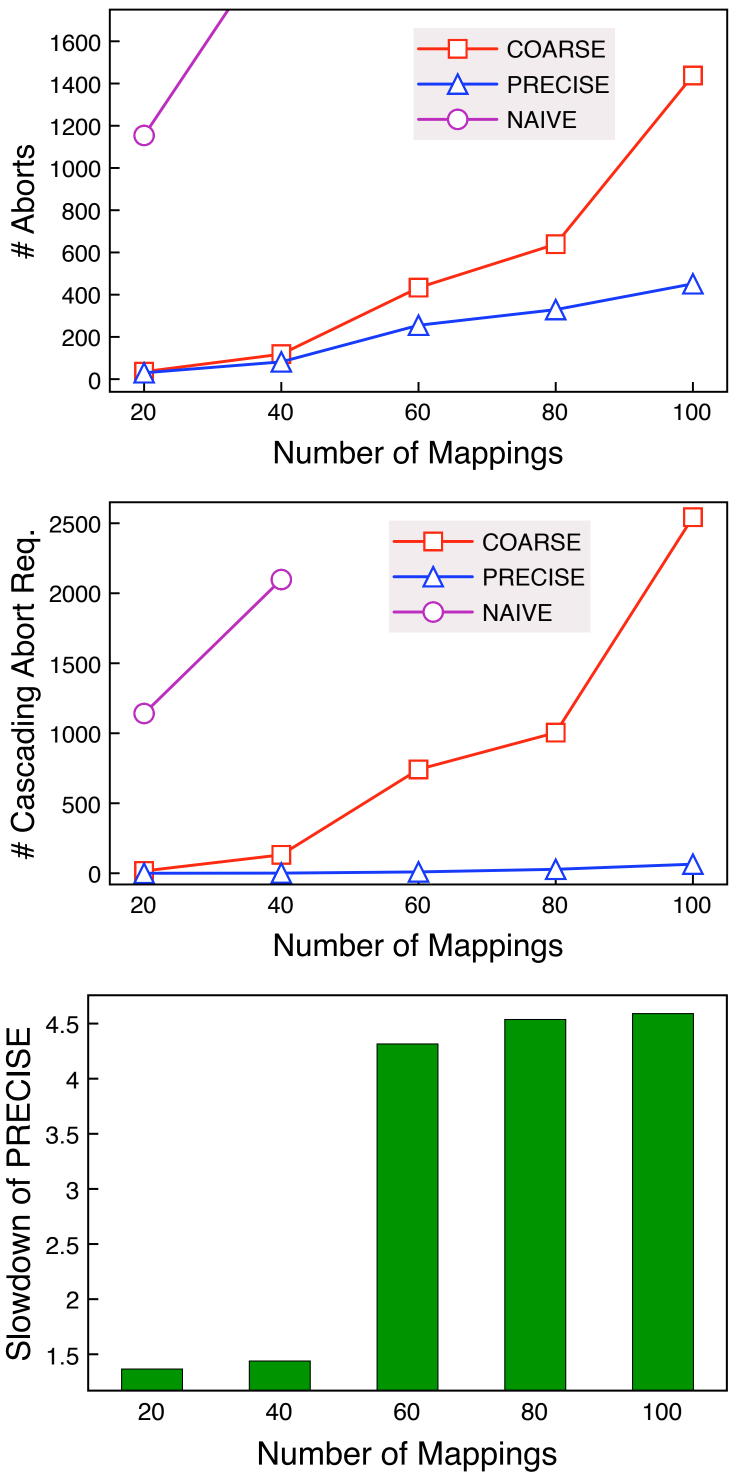

Figure 3.4: Results for the mixed workload

the meantime, abort information related to various writes performed by a chase step is collected and collated. Transactions are frequently marked for abortion multiple times during that phase; however, the scheduler only performs aborts based on the consolidated information. This metric clearly shows the difference between `COARSE` and `PRECISE`; indeed, in scenarios with lower tgd density, `PRECISE` requests *no* cascading aborts.

The final figure shows the relative time penalty associated with the use of `PRECISE` over `COARSE`. This is the ratio of per-transaction execution times for each algorithm. The per-transaction execution time is obtained by dividing the total time for the run by the number of transactions which actually ran (i.e., the original 500 plus the number of aborts). In this way, we adjust for the fact that runs with `PRECISE` involve a lower number of total transaction executions. The graph shows the relative slowdown rather than the per-transaction execution times themselves. The actual execution time increases for each algorithm with the number of tgds, which is not surprising, since more tgds require more read queries and more writes to be performed. Our purpose here is orthogonal to demonstrating this increase: we aim to show the overhead of using `PRECISE` instead of `COARSE` across a variety of tgd density settings.

Our experiments show that `COARSE` and `PRECISE` significantly alleviate the cascading abort problem. As expected, `PRECISE` does best, but at the cost of an increase in execution time. In practice, we expect that the reduction in the number of aborts will be so important to Youtopia users that the increased execution time of `PRECISE` will be acceptable. However, if this time overhead should prove too large, it is also possible to use a hybrid policy combining `COARSE` and `PRECISE` on a per-transaction basis. A transaction which is particularly important and which should not be aborted spuriously – perhaps because it has already aborted several times – can have its read dependencies determined using `PRECISE`, so that it only aborts when it absolutely needs to. For less important transaction, `COARSE` can be used.

We also remark that the absolute number of aborts across all our experiments remains quite high; this underscores the need for a good high-level scheduling policy to minimize the number of aborts that are non-cascading, i.e., due to genuine conflicts.

CHAPTER 4

BEYOND SERIALIZABILITY

In Chapter 3, we explained how a community DBMS can prevent interference between transactions by enforcing YMCSR. However, this is not always a satisfactory solution. Because only non-blocking schedulers are acceptable, Theorem 3.1.9 implies that there is always a risk that some transactions will have to be aborted when enforcing YMCSR. In many situations – even when the transactions do not require other strong guarantees such as durability and *can* in principle be rolled back – such forced aborts are very undesirable or outright unacceptable. The only way to avoid them or reduce their number is to relax the isolation requirements on the running transactions in some way. There is a design space to be explored that lies between the enforcement of full serializability and the total lack of concurrency control.

One way to reduce the need for heavyweight isolation enforcement mechanisms is to bring the user into the picture. It is possible to build a system where a variant of Algorithm 8 would be used only to *detect* conflicts, not to correct them. The correction burden would be shifted to the user, who would examine the system's warning and decide whether the transactions involved required any manual intervention. If the user decided that a transaction did not require an abort despite a conflict, then the system would take their word for it. This approach could potentially work well in some scenarios, where the users are very knowledgeable about the data and transactions, but probably not in the general case. Such "manual override" functionality should certainly always be *available* in a community DBMS, but it is unrealistic to expect that the system could rely on it entirely, for all conflict reconciliation.

What makes more practical sense is for the system to come with a set of default policies for handling transaction interference. In traditional DBMSs, there exist standard *isolation levels* [16] that allow transactions to interleave in a manner which is “somewhat risky” from a concurrency control standpoint, in order to improve performance. These isolation levels are well-known and frequently used in practice, suggesting that even in OLTP-type systems with more traditional workloads and (perhaps) stronger consistency requirements, relaxing serializability for performance reasons is not an outlandish idea.

Developing appropriate isolation levels for Youtopia is ongoing work; in this chapter, we present a high-level discussion of the main challenges that arise when designing such levels for a collaborative DBMS. We also make a concrete proposal for three isolation levels that address some of the challenges we identify.

4.1 Designing isolation levels for a collaborative DBMS

Any isolation level for a transaction is fundamentally a compromise between two sets of constraints. The first set contains more or less explicit desiderata for “correct” transaction execution. The second specifies the concurrency control enforcement mechanisms which are acceptable (or not). As enforcing stronger correctness properties usually requires more heavyweight mechanisms, it is normally impossible to satisfy the constraints in both sets simultaneously. A hierarchy of isolation levels, such as that in [16] or that provided by SQL implementations, defines multiple meaningful tradeoff points with respect to what can and cannot be satisfied.

We note that not all isolation level definitions are explicitly connected to enforcement mechanisms, and indeed the authors of [16] argue convincingly that isolation level definitions should be implementation-independent. Instead, isolation levels can be defined in terms of a set of pathological phenomena: each level in the hierarchy specifies which of the phenomena will be prohibited. Still, in any definition geared towards practical usage (rather than, say, purely theoretical investigation of schedule properties) the choice of the specific set of pathological phenomena to be avoided is directly motivated by the constraints on the acceptability of enforcement mechanisms. For example, as we have seen in Theorem 3.1.9, if blocking schedulers are not acceptable, then forced transaction aborts – a clear example of a pathological phenomenon – *cannot* be avoided (at least when attempting to enforce a nontrivial property of schedules). Thus, while isolation level definitions should definitely be as implementation-independent as possible, they are very closely tied to the constraints on the class of algorithms and mechanisms to be used for enforcement. Consequently, the latter set of constraints must always be kept in mind when designing isolation levels for a new system model such as a collaborative DBMS.

Which properties are desirable for transaction execution in a collaborative DBMS, and which enforcement mechanisms are acceptable or unacceptable? We begin with the first question. We have already discussed transaction serializability. Although the specific kind of serializability desired need not always be YMCSR, it is clear that users will frequently wish to maintain the illusion of serial transaction execution.

Another desideratum relates to abort handling. Enforcing YMCSR, as we explained in Chapter 3, requires post-abort "cleanup" that removes from the system all traces of the writes of the aborted transactions. In the definition of YMCSR, this is made explicit in the requirement that a transaction may not have reads that are affected by a write from an aborted transaction. However, more relaxed versions of this property may be acceptable in some cases. For example, when a transaction aborts, it may be enough to ensure that all *future* reads in the system are unaffected by its writes, while any reads that have already happened are “left alone”. This weaker constraint has the advantage that cascading aborts are not needed to enforce it. As we see, a single high-level desideratum (“aborted transactions should be cleaned up”) does not directly formalize into a single pathological phenomenon, but can instead yield a set of phenomena of which some or all can be prohibited at a given isolation level. *How* a high-level concept is split into a set of such phenomena is directly determined by what is enforceable by the various classes of acceptable mechanisms.

A third property of transaction execution which we have repeatedly mentioned, but not yet addressed, is guaranteeing durability for transactions that perform a *commit*. As in classical OLTP, if a transaction actually commits, it should never have to be rolled back and redone.

What about the constraints on enforcement mechanisms in a community DBMS? In Chapter 3, we discussed at length the importance of non-blocking enforcement mechanisms and the undesirability of aborts that are forced by the system. The versioning mechanism used in Chapter 3 may also be something that we wish to drop. Even if there are no reasons not to have it in place, it is instructive to understand what properties remain enforceable when it is removed.

It is interesting to examine the similarities and differences between the community DBMS desiderata and constraints which we have just presented, and those found in traditional OLTP systems. Such a comparison is educational in itself, but it also demonstrates that the standard isolation levels found in the literature are unlikely to be directly applicable to a collaborative DBMS. Moreover, as we will see, the inapplicability of traditional isolation levels is fundamental – it occurs for reasons beyond the technical fact that the levels' definitions do not carry over to our setting directly.

Our enforcement mechanism constraints are not very unusual, and they would not be unreasonable in an OLTP system. The only significant feature unique to our setting is the *absolute* unacceptability of blocking schedulers. With respect to our three desirable properties of transaction execution, the situation is more complicated. Serializability is the most similar one, although there is the issue of non-terminating transactions that must be dealt with. The statement of 3.2.9 does, however, demonstrate how to relate our serializability desideratum to the classical one – we desire that in all those cases where transactions *do* terminate, classical serializability (or an appropriate relaxation) should hold.

Post-abort cleanup issues are next. In this case, there is a significant difference between the collaborative DBMS and classical settings: relaxing the post-abort cleanup requirement is a very unusual thing to do in classical concurrency control. In a collaborative DBMS, users are likely to be wiling to compromise a lot more with respect to this particular guarantee.

The same is true for our third property – in the classical database world, all transactions commit, and relaxing durability in any way is normally not acceptable at all. This is why issues of durability are not even explicitly addressed when discussing classical isolation levels. (Nonetheless, they do arise in discussions of schedule recoverability, and some protocols – such as strict two-phase locking – actually happen to do "double duty" and enforce both isolation levels and recoverability properties). In a community DBMS, on the other hand, dropping or relaxing durability might be quite reasonable. This is quite fortunate, since – as we have already mentioned – Theorem 3.1.9 implies that a nonblocking scheduler cannot enforce both durability and YMCSR or any other nontrivial property of schedules.

A very large set of community DBMS use scenarios does not require durability; for example, our travel database scenario can be expected to function perfectly well without it. Wikipedia certainly provides no durability for any (basic editing) operation, but this does not stop it from being a very highly successful system that is useful to millions of people. Another option acceptable in a community DBMS but not in a classical one might introduce a limited form of durability where a transaction's writes persist post-commit, but the *isolation level* at which it committed may retroactively change. Consider the library book order example (Example 1.3.4), and suppose that the order transaction does commit and the order is placed. If some data does subsequently change and retroactively affect the order transaction's reads, the library employees would not necessarily cancel the order and redo it. Instead, they might well allow it to go ahead anyway. This real-world human decision is precisely an informal version of dropping the isolation level of the book order transaction post-commit.

4.2 Three isolation levels for Youtopia

In this section, we present the first stages of our more concrete work on Youtopia isolation levels. We present three levels that lie along a spectrum whose endpoints are YMCSR and absolute lack of concurrency control enforcement. Each level serves to illustrate a well-motivated tradeoff point between the constraints on transaction execution properties and on enforcement mechanisms which we have just discussed. The levels form a hierarchy; if a transaction runs at level i, it also runs at level j for all $j > i$.

The presentation of the isolation levels assumes our Youtopia transaction model and the versioned setting from Chapter 3. It also continues to also assume that no transactions ever perform a *commit* operation.

The most restrictive level is based on YMCSR:

Definition 4.2.1 (Isolation Level 1)**.** *Let (D, τ) be a multiversion schedule. A transaction number i in the schedule runs at Isolation Level* 1 *(*serializable*) iff:*

- *it performs no reads that are retroactively affected by a writes of a transaction j, with $j < i$*
- *it performs no reads that are affected by a write by a transaction j which performs an* abort *operation*

The first bullet point in the definition can be enforced by our conflict test from Definition 3.2.5, and the second by appropriately "cleaning up" after an aborted transaction (removing its writes and aborting any transactions in its RF^* relation). The following lemma is clear:

Lemma 4.2.2. *A schedule (D, τ) is in YMCSR iff all the transactions in the schedule run at isolation level* 1*.*

The first obvious relaxation from level 1 is obtained by realizing that the expensive part of enforcing it is our choice to detect and correct the "premature" reads. Retroactively affected reads are expensive to identify, and when they are found, the affected transaction(s) must abort. Dropping this check yields the next isolation level:

Definition 4.2.3 (Isolation Level 2). *Let (D, τ) be a multiversion schedule. A transaction number i in the schedule runs at Isolation Level* 2 *(*unsafe reads permitted*) iff:*

- *it performs no reads that are affected by a write by a transaction j which performs an* abort *operation*

Isolation level 2 is a substantial relaxation from level 1. It is also related, though not directly comparable to, multiversion isolation levels such as snapshot isolation [18, 62]. It is interesting to consider whether a notion of snapshot isolation modified to handle our intensionally specified reads might be useful as well. Our Isolation Level 2 makes the most recent version of each tuple visible to reads. In a snapshot-isolation-like setting, on the other hand, each transaction would only see the most recent version of each tuple that existed at the time the transaction started. This in itself would be easy to implement; however, intuition suggests that in a community DBMS, where transactions may be very long-running, it is better to make all available writes visible to a running transaction rather than forcing it to read a potentially stale database. Moreover, the second part of the standard snapshot isolation definition requires that transactions' write sets be disjoint – this is likely to be more problematic in our setting, as it is unclear when enforcement of this property should occur for long-running transactions.

Enforcing Isolation Level 2 still requires cascading aborts. If no cascading aborts are permitted, there is a further relaxation that can be used instead:

Definition 4.2.4 (Isolation Level 3). *Let (D, τ) be a schedule. A transaction number i in the schedule runs at Isolation Level* 3 *(*read from non-aborted*) iff the following is true. Suppose the transaction* does *perform a read that is affected by a* *write by a transaction j which performs an* abort *operation.* The read must occur prior to the abort.

The reason that Isolation Level 3 can actually be enforced – globally, for all transactions in the system – without requiring aborts is given by Theorem 3.1.9. In our multiversion setting, Isolation Level 3 is actually (associated with) a trivial property of schedules, and such properties can be enforced strictly. The triviality is due to our abort cleanup mechanism which is part of the system model; as soon as a transaction aborts, its tuple versions become invisible to any future reads automatically. Thus it is impossible for a read that happens post-abort to be affected by any tuple version created by the aborted transaction.

On the other hand, if the versioned setting were not in place, enforcing something similar to Isolation Level 3 would become much less "trivial". Intuitively, it is easy to define a conflict between a read and write in a versioned setting, as in Definition 3.2.5, because there is no "overwriting" of versions. It is sensible to postulate that a tuple write (i.e. the creation of a new version) affects the result of a read query on a given database iff the presence or absence of that one tuple version in the database leads to a change in the query result. If overwriting is permitted, however, it is much harder to describe when a write – which may have occurred some time ago and incurred overwriting – affects a subsequent read.

The above observation also suggests that some other tradeoff points between our constraints might be much harder to formalize and implement. For example, it may be interesting from a pragmatic standpoint to explore that part of the design space where where serializability is not an issue at all, but post-abort cleanup is. That is, it would be acceptable for running transactions to interleave arbitrarily, but any aborted transactions should incur full cleanup, including cascading aborts of any “tainted” transactions. The real-world motivation for this point in the design space is a setting where aborts are considered very serious as they occur only due to problems like vandalism. As such, they should trigger stop-the-world cleanups. On the other hand, transactions that are not flagged as vandalism or otherwise dangerous can and should be allowed to proceed without much concern for their interleaving. Conceptually, an isolation level for this scenario would sit “between” our levels 2 and 3 above. However, defining it precisely would again also require defining a read-write conflict without resorting to the versioning mechanism (which presumably would not be in place if no transaction serializability of any sort were required). Developing such a definitional framework is future work.

CHAPTER 5

RELATED WORK

With the rise of Web 2.0, recent years have seen a tremendous amount of interest in collaborative information management, both from academic communities and from industry. In this chapter, we do not propose to survey all of this recent (and less recent) work. We choose instead to focus on the projects and systems most relevant to collaborative management of *structured* data, and on the literature which provides explicit technical background for the research in this thesis. We also note that the body of the thesis sometimes indicates specific papers or systems when they are directly relevant; the presentation in this chapter does not always duplicate those mentions.

5.1 Tgds, the chase and data integration

We have already mentioned that tgds and equivalent constraints such as GLAV mappings [35, 49] and conjunctive inclusion dependencies [47] are well-known in the data integration literature [39, 30, 42, 45, 63]. Chasing with tgds is a well-studied process in general [14, 50, 25] and in the context of data integration and exchange [30]. Recent years have seen new theoretical work investigating issues of chase termination [52] and dealing with nonterminating chases for the purpose of other tasks such as query answering [22]. We note that the authors of [52] also present a method for executing a chase in a monitored fashion and controlling or bounding its behavior to prevent nontermination, although their mechanisms are different from the Youtopia frontier operations.

Still regarding the chase, we note that while the names of the two Youtopia chase methods – the forward chase and the backward chase – may suggest a similarity to the *chase and backchase (C & B)* technique [26], there is not a close relationship between C & B and our work. In C & B, the two chases proceed in distinct phases while ours are interleaved; moreover, C & B is a mechanism for query optimization on a given database, while our chases serve to propagate changes to the data.

Beyond individual constraints and formalisms, there is a growing body of work which adapts classical data integration ideas to the community (or peer-to-peer) setting, including substantial theoretical work [36, 23, 34]. This work has resulted in the creation of systems like Orchestra [39], Piazza [59, 42], Hyperion [54] and the system introduced in [45], as well as real-world scientific data sharing portals such as BIRN [2] and GEON [4]. Among those systems, as we have indicated, Orchestra is closest in spirit to Youtopia due to its use of update exchange; however, we do not make the same weak acyclicity assumption on tgds. As future work on Youtopia progresses to include more data management functionality, we anticipate that many techniques developed in the above systems may prove useful. For example, [45] presents methods for enabling users to register new tables with the system, and the Orchestra project has also addressed issues of disagreement management and provenance [60] and physical implementation and storage [61] for community DBMSs.

5.2 Sharing structured data

A large number of systems and solutions, both academic and industrial, explore various aspects of structured data sharing by communities without explicitly positioning themselves as spiritual successors of classical data integration work. This includes community information management projects such as Cimpl/DBLife [28], MOBS [51] and Dataspaces [41], as well as some work which is not explicitly carried out within the context of a community DBMS, but explores relevant issues.

Among the questions investigated by this body of work, we find for example the problem of encouraging high-quality user participation to improve the data repository [27, 24]. Work has also been carried out on making optimal use of this human attention through the use of hybrid human/automated mechanisms [24, 43], as well as through algorithms that can learn cleaning transformations from user operations [15]. (Although the authors of [15] do not explicitly situate their work within a community setting, it is clear that their algorithm could be adapted to this type of system.)

Another challenge is bootstrapping large-scale repositories, whether through automated methods like collecting relational data from the Internet [21], through manual importation of data by users aided by a tool providing powerful and intuitive reformatting operations [53], or through hybrid human/automated methods [24].

Whether the data in the system comes from bootstrapping or human – or algorithmic – insertions, managing information about its provenance [20] and about users' beliefs as to its reliability [38] is an important challenge and has also been explored.

Finally, another relevant question that has been investigated is facilitating data access in a community DBMS. Fusion Tables [6] and Google Base [5] represent significant work in designing intuitive query interfaces. Such interfaces can provide

SQL-like functionality, as in Fusion Tables, or enable drill-down through a combination of keyword queries with navigation on predefined views, as in Google Base. The management of such standing views or queries in a community DBMS has also been investigated in [46].

In the real world, several highly successful systems for sharing relational data exist, mostly in vertical (topic-specific) settings. Two examples are the CourseRank system [48] and the craft site Ravelry [9]. These systems are based – to our knowledge – on shared DBMS technology, but a thorough understanding by the designers of the communities' needs has enabled them to provide very useful data sharing systems. Both sites attract highly motivated users and this allows them to maintain good data quality. Nonetheless, in these systems, the limitations of using a shared DBMS sometimes become apparent in interesting ways. Schema extensibility, for example, is something that is frequently desired even by nontechnical users. For example, recent months have seen a debate on Ravelry about the site's (current) inability to meet users' wishes for tables devoted to additional crafts [33].

5.3 Concurrency control

Concurrency control is a mature field of database research with a very significant associated body of work. An extensive overview can be found in the textbook [62]. Here, we only indicate work of particular relevance to the material presented in Chapters 3 and 4.

Multiversion concurrency control is itself a topic with a rich literature; [19] is a seminal paper, as are [58] and [17]. MCSR specifically was studied in detail in [40]. More references can be found in the bibliography section at the end of

Chapter Five in [62]. Our definition of YMCSR is also related to predicate [29] and precision locking [44], since it brings together intensionally specified reads and extensionally specified writes.

Isolation levels for transactions have been studied by researchers [18, 16, 31, 32] and are a standard feature in modern database systems. Indeed, ANSI-SQL standards contain explicit definitions of the isolation levels to be supported [13].

Finally, while the Youtopia transaction model has many unique features, long-running transactions are not unknown in other kinds of databases. Approaches for dealing with such transactions in the literature include semantics-based solutions like transaction chopping [57] and other methods that decompose transactions into smaller units [37], as well as alternative mechanisms such as altruistic locking [55]. As we continue the investigation begun in Chapter 4, we expect that some of the solutions from that body of work may prove useful to Youtopia transactions as well.

BIBLIOGRAPHY

[1] Assembla. http://www.assembla.com.

[2] BIRN. http://www.nbirn.net.

[3] Geni. http://www.geni.com.

[4] GEON. http://www.geongrid.org.

[5] Google Base. http://base.google.com.

[6] Google Fusion Tables. http://tables.googlelabs.com.

[7] Google Fusion Tables Users' Group. http://groups.google.com/group/fusion-tables-users-group.

[8] Google Wave. http://wave.google.com.

[9] Ravelry. http://www.ravelry.com.

[10] Tripit. http://www.tripit.com/.

[11] Wikipedia help page. http://en.wikipedia.org/wiki/Help:Merging.

[12] Wiktravel.org Europe page. http://wikitravel.org/en/Europe.

[13] *ANSI X3.135-1992, American National Standard for Information Systems Database Language SQL*. 1992.

[14] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Efficient optimization of a class of relational expressions. *ACM Trans. Database Syst.*, 4(4):435–454, 1979.

[15] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. Learning string transformations from examples. *PVLDB*, 2(1):514–525, 2009.

[16] Patrick O'Neil Atul Adya, Barbara Liskov. Generalized isolation level definitions. In *ICDE*, 2000.

[17] Rudolf Bayer, Hans Heller, and Angelika Reiser. Parallelism and recovery in database systems. *ACM Trans. Database Syst.*, 5(2):139–156, 1980.

[18] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD Conference*, pages 1–10, 1995.

[19] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[20] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *SIGMOD Conference*, pages 539–550, 2006.

[21] Michael J. Cafarella, Alon Y. Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. WebTables: exploring the power of tables on the web. *PVLDB*, 1(1):538–549, 2008.

[22] Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. In *KR*, pages 70–80, 2008.

[23] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Logical foundations of peer-to-peer data integration. In *PODS*, pages 241–251, 2004.

[24] Pedro DeRose, Xiaoyong Chai, Byron J. Gao, Warren Shen, AnHai Doan, Philip Bohannon, and Xiaojin Zhu. Building community wikipedias: A machine-human partnership approach. In *ICDE*, pages 646–655, 2008.

[25] Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. The chase revisited. In *PODS*, pages 149–158, 2008.

[26] Alin Deutsch, Lucian Popa, and Val Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB*, pages 459–470, 1999.

[27] AnHai Doan and Robert McCann. Building data integration systems: A mass collaboration approach. In *IIWeb*, pages 183–188, 2003.

[28] AnHai Doan, Raghu Ramakrishnan, Fei Chen, Pedro DeRose, Yoonkyong Lee, Robert McCann, Mayssam Sayyadian, and Warren Shen. Community information management. *IEEE Data Eng. Bull.*, 29(1):64–72, 2006.

[29] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[30] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[31] Alan Fekete. Allocating isolation levels to transactions. In *PODS*, pages 206–215, 2005.

[32] Alan Fekete, Shirley Goldrei, and Jorge Perez Asenjo. Quantifying isolation anomalies. *PVLDB*, 2(1):467–478, 2009.

[33] Casey Forbes. Ravelry unresolved issue 1481, other fiber arts. http://www.ravelry.com/issues/1481.

[34] Enrico Franconi, Gabriel Kuper, Andrei Lopatenko, and Luciano Serafini. A robust logical and computational characterisation of peer-to-peer database systems. In *DBISP2P*, pages 64–76, 2003.

[35] Marc Friedman, Alon Y. Levy, and Todd D. Millstein. Navigational plans for data integration. In *AAAI/IAAI*, pages 67–73, 1999.

[36] Ariel Fuxman, Phokion G. Kolaitis, Renée J. Miller, and Wang Chiew Tan. Peer data exchange. *ACM Trans. Database Syst.*, 31(4):1454–1498, 2006.

[37] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD Conference*, pages 249–259, 1987.

[38] Wolfgang Gatterbauer, Magdalena Balazinska, Nodira Khoussainova, and Dan Suciu. Believe it or not: Adding belief annotations to databases. *PVLDB*, 2(1):1–12, 2009.

[39] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *VLDB*, pages 675–686, 2007.

[40] Thanasis Hadzilacos and Christos H. Papadimitriou. Algorithmic aspects of multiversion concurrency control. *J. Comput. Syst. Sci.*, 33(2):297–310, 1986.

[41] Alon Y. Halevy, Michael J. Franklin, and David Maier. Principles of dataspace systems. In *PODS*, pages 1–9, 2006.

[42] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema mediation for large-scale semantic data sharing. *VLDB J.*, 14(1):68–83, 2005.

[43] Shawn R. Jeffery, Michael J. Franklin, and Alon Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD*, pages 847–860, 2008.

[44] J. R. Jordan, J. Banerjee, and R. B. Batman. Precision locks. In *SIGMOD*, pages 143–147, 1981.

[45] Yannis Katsis, Alin Deutsch, and Yannis Papakonstantinou. Interactive source registration in community-oriented information integration. In *VLDB*, 2008.

[46] Nodira Khoussainova, Magdalena Balazinska, Wolfgang Gatterbauer, YongChul Kwon, and Dan Suciu. A case for a collaborative query management system. In *CIDR*, 2009.

[47] Christoph Koch. Query rewriting with symmetric constraints. *AI Commun.*, 17(2):41–55, 2004.

[48] Georgia Koutrika, Benjamin Bercovitz, Robert Ikeda, Filip Kaliszan, Henry Liou, Zahra Mohammadi Zadeh, and Hector Garcia-Molina. Social systems: Can we do more than just poke friends? In *CIDR*, 2009.

[49] Jayant Madhavan and Alon Y. Halevy. Composing mappings among data sources. In *VLDB*, pages 572–583, 2003.

[50] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM Trans. Database Syst.*, 4(4):455–469, 1979.

[51] Robert McCann, Alexander Kramnik, Warren Shen, Vanitha Varadarajan, Olu Sobulo, and AnHai Doan. Integrating data from disparate sources: A mass collaboration approach. In *ICDE*, pages 487–488, 2005.

[52] Michael Meier, Michael Schmidt, and Georg Lausen. On chase termination beyond stratification. *PVLDB*, 2(1):970–981, 2009.

[53] Vijayshankar Raman and Joseph M. Hellerstein. Potter's Wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.

[54] Patricia Rodríguez-Gianolli, Maddalena Garzetti, Lei Jiang, Anastasios Kementsietsidis, Iluju Kiringa, Mehedi Masud, Renée J. Miller, and John Mylopoulos. Data sharing in the Hyperion peer database system. In *VLDB*, pages 1291–1294, 2005.

[55] Kenneth Salem, Hector Garcia-Molina, and Jeannie Shands. Altruistic locking. *ACM Trans. Database Syst.*, 19(1):117–165, 1994.

[56] Anish Das Sarma, Xin Dong, and Alon Y. Halevy. Bootstrapping pay-as-you-go data integration systems. In *SIGMOD*, pages 861–874, 2008.

[57] Dennis Shasha, François Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst.*, 20(3):325–363, 1995.

[58] Richard Edwin Stearns and Daniel J. Rosenkrantz. Distributed database concurrency controls using before-values. In *SIGMOD Conference*, pages 74–83, 1981.

[59] Igor Tatarinov and Alon Y. Halevy. Efficient query reformulation in peer-data management systems. In *SIGMOD Conference*, pages 539–550, 2004.

[60] Nicholas E. Taylor and Zachary G. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD*, pages 13–24, 2006.

[61] Nicholas E. Taylor and Zachary G. Ives. Reliable storage and querying for collaborative data sharing systems. In *ICDE*, 2010.

[62] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.

[63] Ling-Ling Yan, Renée J. Miller, Laura M. Haas, and Ronald Fagin. Data-driven understanding and refinement of schema mappings. In *SIGMOD*, pages 485–496, 2001.