LAZY TRANSACTION EXECUTION MODELS

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

Sudip Roy

January 2015

LAZY TRANSACTION EXECUTION MODELS

Sudip Roy, Ph.D.

Cornell University 2015

Transactions are the fundamental unit of change as perceived by the database. In Online Transaction Processing (OLTP) applications, transaction behavior is often insensitive to the intial database state. For example, in a flight booking application, a user may not care what exact seat is booked as long as it is a window seat. Similarly, in an online shopping application, the exact value of the stock level of an item is irrelevant for a purchase transaction as long as there is enough stock to fulfill an order. Such insensitivity of transactions towards the initial database state presents opportunities for optimizing system performance.

In this dissertation, we present two systems which exploit such insensitivity in transactions by deferring the execution of certain operations until the effect of such operations are externally perceivable. First we present Quantum Databases — a lazy transaction processing system that defers the making of choices in transactions until an application or user forces the choice by observation. Conceptually, the database is in a quantum state — in one of many possible worlds, exactly which one is unknown — until fixed by observation. Next, we present Homeostasis — a lazy transaction processing system for distributed or replicated databases which automatically identifies insensitivity of a set of transactions towards the database state and exploits it to minimize the amount of inter-node communication required to guarantee consistency. The key insight is to defer the synchronization of distributed state until such laziness affects the behavior of transactions.

BIOGRAPHICAL SKETCH

Sudip Roy was born and brought up in the state of Odisha in India. He completed his higher secondary education from B.J.B. Junior College, Bhubaneswar, and secured the top rank in the annual higher secondary examination in state of Odisha.

He then attended Indian Institute of Technology(IIT), Kharagpur from 2005 to 2009, and graduated with a B. Tech. in Computer Science and Engineering. While at IIT, he worked with Prof. Partha Pratim Chakrabarti on Qualitative Reasoning for which he was awarded the best undergraduate thesis award. During his stint at IIT, he received a number of fellowships and graduated with the second highest GPA in the class of 2009.

Sudip then joined Cornell University where he was advised by Prof. Johannes Gehrke. During his PhD he worked on various aspects of transaction processing systems including designing new abstractions for supporting coordination in databases, lazy transaction execution models for resource allocation applications and semantic-based adaptive data consistency for wide area transaction processing systems. During his PhD, Sudip received the Auto Desk fellowship, a SIGMOD best paper award and a Yahoo! Graduate Teaching award. Apart from academic research, Sudip's internship work at Google Research and Microsoft Research have influenced products and spurred new directions of research.

After graduation, Sudip will be joining the Structured Data Group at Google Research as a Research Scientist.

To my family who believe in me, more than I believe in myself.

ACKNOWLEDGEMENTS

Completing my PhD under the guidance of my advisor, Prof. Johannes Gehrke, has been a privilege. I would like to thank Johannes for giving me a great degree of freedom in choosing my research topics, teaching me the basics of doing good research, helping me when I got stuck, motivating me when I felt low, and above all, guiding me to establish myself as an independent researcher. He has and will always be a source of inspiration.

I was fortunate to be have had the opportunity to work with Prof. Christoph Koch and Lucja Kot during my PhD. They have helped me in distilling my ideas into more elegant formalism, and have taught me the art of presenting technical content in an interesting, unambiguous and coherent way.

I would like to thank my internship mentors and collaborators who often provided a different perspective on various aspects of research and life as a researcher. Special thanks to Christian König and Jayant Madhavan for not only giving me insights into research in the industry but also helping me in deciding a suitable career path.

I would like to thank members of my committee, Prof. Andrew Myers, Prof. Joe Halpern and Prof. Levent Orman for providing valuable comments, for pointing me to related literature from each of their communities, and finally, for their encouraging words which helped me enhance the contributions of this dissertation.

I am also thankful to all my friends with whom I had the pleasure to work with and engage in, often long, discussions about the trials and tribulations of the PhD journey.

Finally, I would like to thank my parents, brother and sister-in-law for being an unending source of inspiration and providing unconditional support

throughout my life. Last but not the least I would like to thank Gargi, the love of my life, for silently tolerating my grumpiness, being incredibly supportive, and believing in me through the highs and lows of the last year of PhD.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

What we observe is not nature itself,
but nature exposed to our method of questioning.

— Werner Heisenberg

Concurrent changes to the shared state of a system need protection mechanisms to prevent interference. Different communities have developed abstractions and primitives to make concurrent changes to the shared state without corrupting it. In the database community, this work has culminated in the concept of a *transaction*. A transaction is a discrete unit of work performed over a shared database as reflected in the ACID properties of atomicity, consistency, isolation, and durability: it provides the conceptual properties of executing completely or not at all, of preserving database consistency as it runs, of running without interference from other transactions, and if completed, of making its changes persistent.

Database systems which provide support for storing and updating application state transactionally are typically referred to as transaction processing systems. Such transaction processing systems strive to maximize transactional throughput and minimize execution latency while ensuring correctness as defined by the ACID properties. Transaction processing systems which require near real time execution latency are often referred to as Online Transaction Processing (OLTP) systems. Over recent years increasingly demanding require-

ments from applications have often run into conflict with ACID properties and mechanisms for enforcing such properties in OLTP systems. One such example is data-driven coordination [57] where information flow among transactions is necessary for completing application tasks which are inherently collaborative. Such information flow is in fundamental conflict with the isolation guarantees of traditional transactions. While *entangled queries* [46] and *entangled transactions* [47] have been proposed as clean and powerful abstractions for expressing such communication between transactions, existing transaction execution models are ill-suited for supporting any form of communication between concurrent transactions. A second such example fast *and* consistent datastores that are either distributed or replicated across a wide area. While protocols that provide strong consistency over such wide-area distributed systems have unacceptably high transaction latency, eventually-consistent systems sacrifice strong consistency to achieve low response times, and thereby expose a non-intuitive programming interface [11]. In this dissertation, we propose transaction execution models which enhance and extend existing OLTP systems to support such requirements for a large class of modern OLTP applications.

The key idea explored in this dissertation is to delay the execution of operations in transactions until they are externally perceivable. In the context of databases, such external perception is determined by the reads performed by the transactions and the actions taken within the transaction based on such reads. In this dissertation, we observe that by understanding the semantics of the programs executed transactionally, we can often delay the execution of certain operations in transactions until a suitable point in the future. Lazy execution of such "imminently imperceptible" transaction operations can, for large classes of applications, improve system performance by increasing throughput

and reducing execution latency. Specifically, we present *lazy transaction execution models* for two classes of OLTP applications – first, for applications that use transactions to allocate resources to users, and second, for applications which use transactions to update their state distributed(or replicated) over a wide area. For the first class of applications, by *lazily binding* particular resources to users, it is possible to make better and more informed decisions which maximize some notion of global utility for the application. For the second class of applications, we delay communicating changes to the database made by transactions at a particular site of the distributed database so long as such delay does not influence the behavior of transactions executing at other sites. Such *lazy synchronization* of distributed state reduces execution latency by removing communication overhead from the critical path of transaction execution. In the rest of this chapter, we first give examples of the opportunities for lazy execution in the context of the two classes of applications, and then present an overview of our proposed solution and of the contributions of this dissertation.

1.1 Lazy Binding

Database applications are often used to allocate commodities or resources based on user requests. These commodities could be physical goods in retail applications, virtual goods in a game, the availability of shared-use physical or virtual resources such as a hotel room or an Amazon EC2 instance, time slots on a calendar, or even another human user, e.g., a carpool partner. Two aspects that of resource allocation are interesting. First, resource allocation comes with conditions from users. For example, travelers specify whether they want an aisle or window seat, employees scheduling meetings specify whether they want a

large or small meeting room, and so on. However, there is often flexibility on other features of the requested resource. Second, typically there is some delay between user requests and the actual "consumption" of the resource in question. For example, travelers reserve plane tickets some time in advance and there is time on the order of days — or months — between the time the booking transaction commits and the time the seat is actually used.

We observe that both user flexibility and request-to-usage time delay can and should be exploited by the system for optimal resource allocation. The system can exploit user flexibility by explicitly keeping track of preferences and "don't cares"; it can also exploit the delay by deferring resource allocation as late as possible.

Usually, requests for resources arrive over time from different users and the system has no knowledge of the future request sequence. In the absence of such knowledge, allocating a resource too early may prevent other requests from being fulfilled; for example, if a traveler – Mickey – who does not care about his seat is assigned the last available window seat on a plane, a subsequent user who only wants a window seat may be turned away, unless Mickey is willing to be reseated to an aisle seat. Reseating Mickey could be nontrivial and may require executing compensation logic; for example, if he is traveling with his family and they all want to sit in the same row, several people may need to be reseated.

Deferral of resource assignment can not only improve global utility, but the individual user's experience as well. For example, Mickey might have a preference for flying on Delta, but if there are no such available flights, he is willing to fly on any other airlines. It is possible that there are no seats available on Delta

when Mickey submits his transaction; however, a seat may open up due to a future cancellation. If Mickey's flight plans have not been finalized yet, the system can automatically "reassign" Mickey's seat at this point. Moreover, users may specify more sophisticated requests that involve coordination with other users. Formalisms such as entangled and enmeshed queries [46, 21] make it possible for Mickey to specify requests such as "I want to sit next to my friend Goofy" (who will be booking his seat separately). If Goofy's seat request does not arrive in the system until much later, it is again desirable to defer seat assignment to maximize the chance that Mickey and Goofy can sit together.

Calendar management is another application domain where resource allocation is challenging. Users who schedule meetings often have flexibility in their respective calendars, but current software forces everyone to commit to a particular date and time, even if the meeting is months away. Anecdotal evidence suggests that such arbitrarily chosen time slots frequently end up conflicting with higher-priority meetings scheduled at short-notice. For example, suppose Mickey schedules a work offsite with his team two months in advance for a Friday afternoon and everyone on the team makes sure to keep that time slot free. Now, suppose that on the Wednesday before the offsite, Mickey is notified that he needs to join a high-priority meeting with the company CEO on Friday afternoon. Mickey now needs to reschedule the offsite, taking into account the availability of all his team members; by now, it may be impossible to find a new slot in the immediate future when everyone is available at the same time. Rescheduling under such circumstances is time-consuming and often stressful; indeed, many companies employ full-time administrative assistants who devote significant amount of time specifically to calendar management. On the other hand, suppose that Mickey and his team were all willing

to delay finalizing their daily work schedules until the evening of the previous day; high-priority events could now be added at short notice with much less disruption. The uncertainty of not knowing your schedule until the day before may be worrisome to certain employees, but in highly dynamic environments where rescheduling is frequent, employees cannot be certain of their schedules anyway, so it is likely that they would accept our proposed alternative.

Whether in travel planning, calendar management, or other application domains, most of today's resource allocation solutions do not fully utilize the opportunities provided by user flexibility and the request-to-usage time delay. If they do, they achieve this through custom ad-hoc code. There is currently no clean, general solution to the entire class of resource allocation problems, and developing such a solution on top of an existing DBMS is not trivial. First, a user request for a resource that contains preferences does not directly translate into an SQL query that returns a single resource instance, unless one uses an ad-hoc solution like appending `LIMIT 1` to the query. Second, normal database transactions cannot commit without a concrete value being assigned, so deferred assignment through lazy binding is not possible. The typical solution used today is to allocate an ad-hoc placeholder value and change the assignment later as needed, with lots of error-prone logic at the middle tier. Third, databases do not come with functionality for keeping track of user conditions such as seat type requirements or preferences. Therefore, these must be serialized and saved for future use; again, this can be done in an ad-hoc fashion at the application layer but a clean and general solution is desirable. We further discuss the trade-offs between implementing this logic in the middle tier on top of a DBMS versus implementing it within the DBMS itself in Section 2.5.

1.2 Lazy Synchronization

Modern datastores are huge systems that exploit distribution and replication to achieve high availability and enhanced fault tolerance at scale. In many applications, it is important for the datastore to maintain global consistency to guarantee correctness. For example, in systems with replicated data, the various replicas must be kept in sync to provide correct answers to queries. More generally, distributed systems may have global consistency requirements that span multiple nodes. However, maintaining consistency requires coordinating the nodes, and the resulting communication costs can increase transaction latency by an order of magnitude or more [106].

Today, most systems deal with this tradeoff in one of two ways. A popular option is to bias toward high availability and low latency, and propagate updates asynchronously. Unfortunately this approach can only provide eventual consistency guarantees, so applications must either deploy additional mechanisms such as compensations or custom conflict resolution strategies [100, 31], or they must restrict the programming model to eliminate the possibility of conflicts [6]. Another option is to insist on strong consistency, as in Spanner [28], Mesa [48] or PNUTS [27], and accept slower response times due to the use of heavyweight concurrency control protocols.

This dissertation argues for a different approach. Instead of accepting a trade-off between responsiveness and consistency, we demonstrate that by carefully analyzing applications, it is possible to achieve the best of both worlds: strong consistency and low latency in the common case. The key idea is to exploit the semantics of the transactions involved in the execution of an appli-

cation in a way that is safe and completely transparent to programmers.

It is well known that strong consistency is not always required to execute transactions correctly [58, 106], and this insight has been exploited in protocols that allow transactions to operate on slightly stale replicas as long as the staleness is "not enough to affect correctness" [14, 106]. This work takes this basic idea much further, and develops mechanisms for automatically extracting safety predicates from application source code. Our *homeostasis protocol* uses these predicates to allow sites to operate without communicating, as long as any inconsistency is appropriately governed. Unlike prior work, our approach is fully automated and does not require programmers to provide any information about the semantics of transactions.

Example: top-k query To illustrate the key ideas behind our approach in further detail, consider a top-k query over a distributed datastore, as illustrated in Figure 1.1. For simplicity we will consider the case where $k = 2$. This system consists of a number of *item sites* that each maintain a collection of $(key, value)$ pairs that could represent data such as airline reservations or customer purchases. An *aggregator site* maintains a list of top-k items sorted in descending order by *value*. Each item site periodically receives new insertions, and the aggregator site updates the top-k list as needed.

A simple algorithm that implements the top-k query is to have each item site communicate new insertions to the aggregator site, which inserts them into the current top-k list in order, and removes the smallest element of the list. However, every insertion requires a communication round with the aggregator site, even if most of the inserts are for objects not in the top-k. A better idea is to

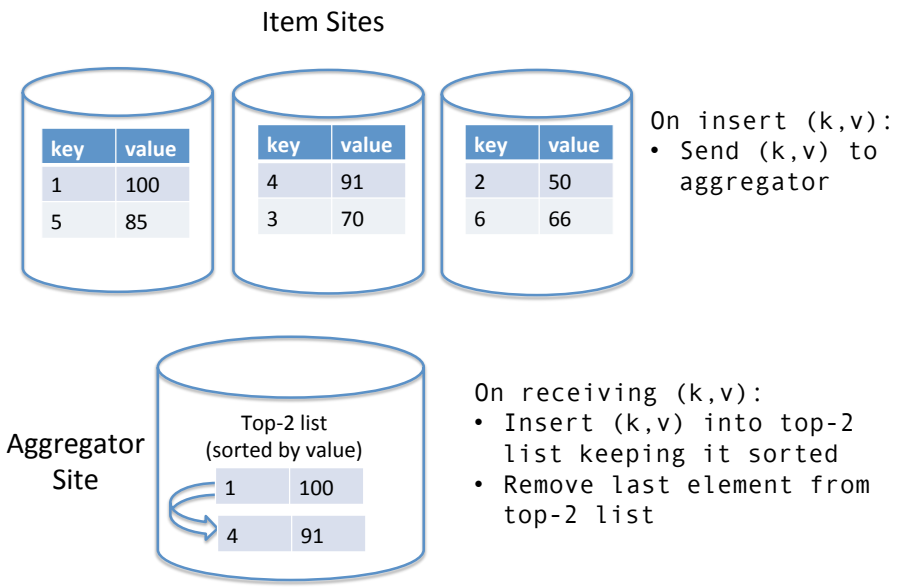

Figure 1.1: Distributed top-2 computation, basic algorithm.

only communicate with the aggregator node if the new *value* is greater than the minimal value of the current top-k list. Each site can maintain a cached value of the smallest value in the top-k and only notify the aggregator site if an item with a larger value is inserted into its local state. This algorithm is illustrated in Figure 1.2, where each item site has a variable *min* with the current lowest top-k value. In expectation, most item inserts do not affect the aggregator's behavior, and consequently, it is safe for them to remain unobserved by the aggregator site—like the proverbial trees that fall in the forest.

This improved top-k algorithm is essentially a simplified distributed version of the well-known *threshold algorithm* for top-k computation [34]. However, note that this algorithm can be extracted *automatically* by analyzing the code for the aggregator site. In Chapter 3, we will show how.

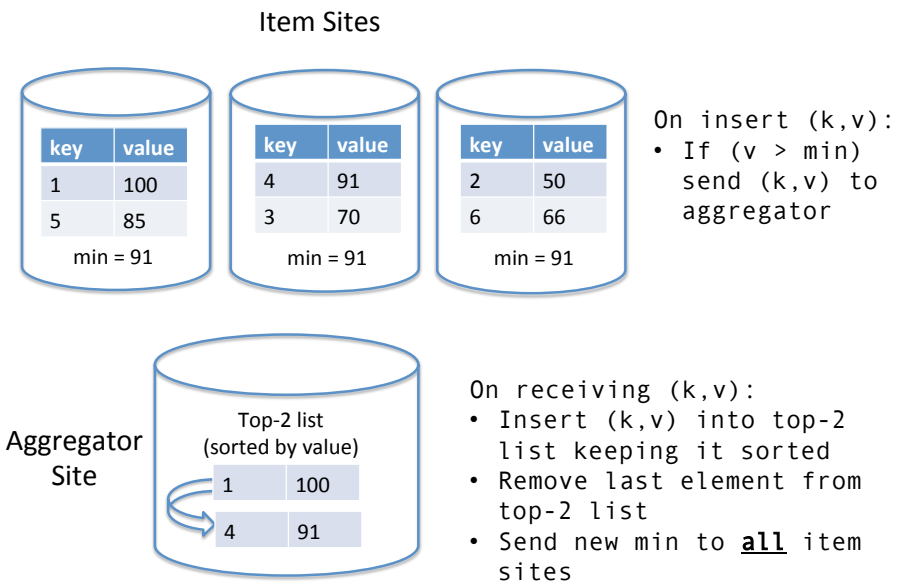

Figure 1.2: Distributed top-2 computation, improved algorithm.

1.3 Outline and Contributions of Dissertation

The work presented in this dissertation is organized in the form of two main chapters; each describing a lazy transaction execution model. In Chapter 2, we first discuss the formal underpinnings for lazy binding of unread values in transactions and then present our system, Quantum Databases, which transparently defers assignment of resources to users for resource allocation applications. In Chapter 3, we present techniques for semantic analysis of transaction programs and propose a new protocol, Homeostasis, which minimizes communication between sites in distributed and replicated databases by deferring and batching updates performed by transactions when such updates are guaranteed not to influence the behavior of other transactions. In Chapter 4, we discuss work from different communities related to the work presented in this dissertation. Finally, we summarize the findings of this dissertation in Chapter 5.

Next, we give a brief overview of the contributions of this dissertation in the

context of each of the two lazy transaction execution models.

1.3.1 Quantum Databases

In Chapter 2, we present our contributions towards developing an end-to-end solution for performing resource allocation reasoning on top of standard relational databases in an OLTP setting. First, we propose *resource transactions* as a formalism that extends SQL to allow the specification of transactions over resources that include user "don't cares" and preferences (Section 2.1).

Second, we introduce *quantum databases* as an abstraction for the associated state that the system maintains as it defers value assignment (Section 2.2). A quantum database allows resource transactions to commit without assigning concrete resource instances; it keeps track of all possible worlds corresponding to all possible concrete resource assignments that *could be made*. A quantum database is an intensional specification of these possible worlds using a set of constraints collected from committed resource transactions. In true quantum fashion, unless the state is observed (i.e., read) by someone, the database remains in all of these states simultaneously. This is similar to probabilistic and uncertain databases [99]; however, the key difference is that uncertainty is strictly internal to the quantum database and a read causes uncertainty to be eliminated, i.e., it forces an instantiation.

Third, we show how a quantum database is maintained as transactions execute in the system. We specify how the database is transformed by operations such as reads, writes, and updates; this involves in some cases reducing the uncertainty and in others introducing more uncertainty. We discuss algorithms

and consistency issues associated with each of these operations, and we propose practical strategies which strike a sweet spot between computational tractability and optimizing resource allocation by keeping the number of possible worlds large (Section 2.2.2).

Fourth, we demonstrate the feasibility of quantum databases by implementing a prototype (Section 2.3) and evaluating it under realistic workloads of a real-world application (Section 2.4).

Lastly, we discuss the research challenges involved in making the quantum database abstraction a real-world tool for developers (Section 2.5). While these challenges are significant, we believe that they are possible to resolve and that working on them will yield new insights into multiple aspects of database systems, well beyond resource allocation applications.

1.3.2 Homeostasis

In Chapter 3, we present our contributions towards developing a new semantics-based method for guaranteeing correct execution of transactions over an inconsistent datastore. Our method is geared towards improving performance in OLTP settings, where the semantics of transactions typically involve a shared global quantity (such as product stock level), most transactions make small incremental changes to this quantity, and transaction behavior is not sensitive to small variations in this quantity except in boundary cases (e.g. the stock level dips below zero). However, our method will still work correctly with workloads that do not satisfy these properties.

We make contributions at two levels. First, we introduce our solution as a high-level framework and protocol. Second, we give concrete instantiations and implementations for each component in the framework, thus providing one particular end-to-end, proof-of-concept system that implements our solution. We stress that the design choices we made in the implementation are not the only possible ones, and that the internals of each component within the overall framework can be refined and/or modified if desired.

The first step in our approach is to analyze the transaction code used in the application (we assume all such code is known up front) to compute a global predicate that tracks its "sensitivity" to different input databases. We then "factorize" the global predicate into a set of local predicates that can be enforced on each node. In the example above, the analysis would determine that the code for the aggregator site leaves the current top-k list unchanged if each newly inserted item is smaller than the current minimal element.

Analyzing transaction code is challenging. It is not obvious what the analysis should compute, although it should be some representation of the transaction semantics that explains how inputs affects outputs. Also, we need to design general analysis algorithms that are not restricted in the same ways as the demarcation protocol [14] or conit-based consistency [106], which are either limited to specific datatypes or require human input.

Our first contribution (Section 3.1) is an analysis technique that describes the semantics of transactions in terms of *symbolic tables*. Given a transaction, a symbolic table is a set of pairs where each pair's first component is a logical predicate on database states, and the second component concisely represents the writes performed by the transaction on databases that satisfy the predicate.

We also show how to compute symbolic tables from transaction code in a simple yet expressive language.

Our second contribution (Section 3.2) is to use symbolic tables to execute transactions while avoiding inter-node communication. Intuitively, we exploit the fact that if the database state does not diverge from the current entry in the symbolic table, then it is safe to make local updates without synchronizing, but if the state diverges, then synchronization is needed. In our top-k example in Figures 1.1 and 1.2, as long as no site receives an item with value higher than the minimal value, no communication is required. On the other hand, if a value greater than the minimal value is inserted at a site, then we would need to notify the aggregator, compute the new top-k list, and communicate the new minimal value to all the sites in the system (including sites that have not received any inserts).

Our *homeostasis protocol* generalizes the example just described and allows disconnected execution while guaranteeing correctness – i.e., that the transactions perform the same writes as during a view-serializable connected execution. It is provably correct and subsumes the demarcation protocol and related approaches. The protocol relies on the enforcement of *global treaties*; these are logical predicates that must hold over the global state of the system. In our example, the global treaty is the statement "the current minimal value in the top-k is 91". If an update to the system state violates the treaty, communication is needed to establish a new treaty.

In general, the global treaty can be a complex formula and checking whether it has been violated may itself require inter-site communication. To avoid this, it is desirable to split the global treaty into a set of *local treaties* that can be checked

at each site and that together imply the global treaty. This splitting can be done in a number of ways, which presents an opportunity for optimization to tune the system to specific workloads. Our third contribution (Section 3.3) is an algorithm for generating local treaties that are chosen based on known workload statistics.

Implementing the homeostasis protocol is nontrivial and raises many systems challenges. Our fourth contribution is a prototype implementation and experimental evaluation (Sections 3.4 and 3.5).

CHAPTER 2

QUANTUM DATABASES

In the strict formulation of the law of causality
– if we know the present, we can calculate the future –
it is not the conclusion that is wrong but the premise.

— Werner Heisenberg

In this chapter, we introduce *quantum databases*, a new database abstraction that allows to defer the making of choices in transactions until an application or user forces the choices by observation. Conceptually, the database is in a quantum state — in one of many possible worlds, which one is unknown — until fixed by observation. Practically, our abstraction enables late binding of values read from the database. This allows more transactions to succeed in environments with high contention. This is particularly important for applications in which transactions compete for scarce physical resources represented by data items in the database, such as seats in airline reservation systems or meeting slots in calendaring systems. In such environments, deferral of the assignment of resources to consumers until all constraints are available to the system will lead to more successful transactions. Through entanglement of queries [46] and transactions [47], quantum databases can enable collaborative applications with a constraint satisfaction aspect directly within the database system.

For concreteness, most of the examples in the chapter are drawn from the travel planning scenario discussed in Section 1.1; however, we also discuss other application domains such as calendar management when relevant.

```
SELECT 'Mickey', F.fno AS @f, F.fdate AS @d,
       A1.sno1 AS @s
FROM Flights F, Available A1,
   OPTIONAL Available A2, OPTIONAL Adjacent J
WHERE
  OPTIONAL ('Goofy', F.fno, F.fdate, A2.sno2)
                                        IN Bookings
  AND Fdest='LA' AND ... -- join condition
CHOOSE 1
FOLLOWED BY (
  DELETE (@f, @d, @s) FROM Available;
  INSERT ('Mickey', @d, @f, @s) INTO Bookings;)
```

Figure 2.1: An example resource transaction

2.1 Resource Transactions

As explained previously in Section 1.1, resource transactions are an extension of SQL allowing users to explicitly specify soft preferences and features that they do not care about in addition to traditional hard constraints. We first introduce the notion of resource transactions through an example and then discuss the salient features of their syntax and semantics.

An example resource transaction is shown in Figure 2.1. This is a transaction that Mickey might issue to specify that he would like one seat on any available flight to LA, and that he has a soft preference for sitting next to Goofy, if Goofy already has a booking on some flight to LA. The `SELECT` and `FROM` clauses are essentially standard SQL; the three new keywords we introduce are `OPTIONAL`, `CHOOSE` and `FOLLOWED BY`. `OPTIONAL` specifies that the specific conjunct in the `WHERE` clause is a soft preference rather than a hard constraint. `CHOOSE 1` specifies that only one seat (tuple) is desired as an answer to the query. Finally, the `FOLLOWED BY` block contains a specification of the database *writes* to be executed based on the result returned by the `SELECT-FROM-WHERE` query. In

this case, the concrete seat chosen is to be deleted from the `Available` table and a suitable `Booking` is to be made.

A resource transaction has two components: first, a query with optional clauses and a `CHOOSE 1` clause, and second, a subsequent code block that involves a set of blind writes to the database. No reads are permitted within the `FOLLOWED BY` block. One might imagine using SQL queries with `OPTIONAL` and `CHOOSE` within more complex code blocks that include subsequent reads and other features. Such an extension to the basic resource transaction model is in principle possible although nontrivial; investigating the practical usefulness of this extension and developing it fully is ongoing work. Ordinary resource transactions as described above are more limited, but nevertheless provide a programming pattern suitable for the majority of our use cases. Requesting a resource and performing a set of blind writes to "reserve" the resource is by far the most common pattern used in resource allocation applications.

In the rest of the chapter we use a Datalog-like notation for representing resource transactions, which is a straightforward equivalent of the SQL representation. Each transaction is denoted as follows:

$$U :-_1 \; B$$

U and B are conjunctions of relational atoms. We call B the body of the transaction, and U the update portion of the transaction. Any variables appearing in U must also appear in B (this is a *range-restriction* requirement). Each atom in U is either a delete (denoted with a leading $-$) or an insert (denoted with a leading $+$) of a single tuple into the database. Optional conditions are underlined and the CHOOSE 1 is denoted by $:-_1$. For example, the intermediate representation for the example in Figure 2.1 is as follows, with `A`, `B`, `Adj`, `M` and `G` abbreviating

Available, Bookings, Adjacent, 'Mickey' and 'Goofy' respectively.

$$-\mathtt{A}(f_1, s_1), +\mathtt{B}(\mathtt{M}, f_1, s_1) :- \begin{cases} \mathtt{A}(f_1, s_1) \wedge \\ \underline{\mathtt{B}(\mathtt{G}, f_1, s_2)} \wedge \underline{\mathtt{Adj}(s_1, s_2)} \end{cases}$$

The body of the transaction contains three atoms. The first specifies that the variable s_1 should be an available seat for Mickey. The two subsequent optional atoms specify that s_2 should be a seat adjacent to s_1 that is already reserved by Goofy. The update portion of the transaction specifies that once a suitable s_1 is found, appropriate changes should be made by deleting a tuple from Available and inserting a tuple into Bookings.

A detailed presentation of the semantics of resource transactions requires formalizing the deferred value assignment model mentioned in the Introduction. As described in detail in Section 2.2, the actual assignment of values to variables and "execution" of the FOLLOWED BY clause happens after the transaction has already committed. In this Section, we give a high-level intuitive overview of the semantics. First, we explain the semantics a resource transaction would have *without* the deferred assignment model, and then explain how deferred assignment both complicates certain things and presents certain unique opportunities.

A system that processes resource transactions without deferred assignment would proceed as follows. First, the body of a resource transaction is *grounded*, i.e., each variable is assigned one specific value from the database. For example, s_1 might be assigned the value 5A, f_1 might be assigned 123 (if 123 is a flight to Los Angeles), and s_2 might be assigned 5B (if this is the seat Goofy has already

booked). We use the term *grounding* and (value) assignment interchangeably in the rest of the chapter. Sometimes it may not be possible to find a value assignment that satisfies all the optional clauses; this is expected and permitted by the semantics, although if there is an assignment that satisfies the optional clauses it must be chosen in preference to one that does not. Once the grounding is finalized, the system makes appropriate changes to the database as specified in the update portion.

Quantum databases implement a variant of the above semantics in a setting where value assignment is not immediate, but rather deferred until after the commit of the transaction. As explained in Section 1.1, instead of grounding and executing the update portion, the system maintains a guarantee that at least one suitable grounding for the committed transaction exists at all times. Once it is necessary or desirable to actually make the updates — for example, once Mickey is at the airport and checking in for his flight — then the system chooses a grounding and performs appropriate database writes. In terms of the programming API, the application is notified of the initial transaction commit only; because of the guarantee that the system subsequently maintains, the transaction will never need to be rolled back and the application is not notified again when the value assignments are actually made. (Such a second notification could in principle be issued if desired, although it is not clear whether this would ever be useful in practice.) The first notification — that the transaction has committed — represents a guarantee that the transaction will achieve its goal of booking a seat when value assignment actually happens.

The deferred grounding execution model has an impact on the "basic" semantics described above. It creates the need for two key design decisions with

regard to the transaction semantics. The first relates to the treatment of optional constraints, and the second to choosing an appropriate notion of transaction serializability.

We begin with the issue of optional constraints. Suppose a committed transaction currently has a possible assignment that satisfies its optional constraints — for example, when Mickey's transaction commits, Goofy has seat `5B` booked and seat `5A` is open for Mickey. Now suppose another transaction arrives and makes a conflicting request. For example, Pluto specifically requests to book seat `5A`, and his constraint is non-optional. Should the system allow Pluto in, or keep `5A` for Mickey? Our design decision is to allow Pluto in, since Mickey's constraint was optional rather than hard. Somewhat more formally, the only invariant that the system maintains for a committed resource transaction is that there exists a satisfying assignment for its *non-optional* body atoms. In fact, optional conditions are checked only when the variables of the transaction are assigned values; if there is an assignment that satisfies optional as well as non-optional atoms, that assignment is chosen.

Second, the notion of serializability becomes interesting when we move to a deferred assignment model. The order in which resource transactions commit is not necessarily the order in which their variable assignments are fixed and in which their database updates are carried out. Suppose the system needs to fix a grounding for a committed resource transaction: in principle, it has two options. It can choose from the values available at the time the resource transaction was committed, or from those available at the time the value must be fixed. These sets may be different; in our running example, we may well expect that the seat availability has changed between the time that Mickey's transaction commit-

ted on Monday and the time that Mickey reads his seat number on Tuesday. Choosing Mickey's seat based on Tuesday's availability is a natural thing to do, but it violates classical transactional isolation. Mickey's transaction is now no longer serialized in commit order. However, the *intent* [36] of his transaction has definitely been preserved, and we have maintained *semantic serializability* as his transaction has achieved all its goals. Maintaining strict classical serializability is also possible; this requires the system to ensure that at least one seat *from those available on Monday* remains available for Mickey. This means the system must be more restrictive in terms of how many other transactions can commit. Quantum databases can implement transactions under either semantic serializability or the classical ACID-style *strong serializability*, although we expect the former to be more natural in most application scenarios.

2.2 Quantum Databases

We now present the details of our execution model for resource transactions which allows for deferred assignment of values to variables in committed transactions. We achieve this by maintaining the database in a partially uncertain state, called a *quantum state*, and updating it appropriately in response to various transactional operations. We call the resulting database a *quantum database.*

2.2.1 Defining a Quantum Database

Consider for a moment an execution model for resource transactions where value assignment is *not* deferred. Rather, at the time the transaction is executed,

suppose the system finds all possible values that could be assigned – all possible flights and seats for Mickey, say – and *forks* the database state into several possible worlds. In the first possible world, Mickey gets assigned the first available seat, in the second he gets the second one, and so on. This yields a large, but finite set of possible worlds, each of which is a concrete database in which Mickey has a concrete seat on a concrete flight.

Suppose we now maintain all these possible worlds explicitly as our database state. Other resource transactions can run on each possible world as well and cause another "forking" of the state; for example, if Donald now submits a transaction, the system can create one possible world for each of Donald's possible seat assignments. If Donald only wanted a window seat, and there was only one available window seat, this would eliminate all worlds in which Mickey got that window seat. In other words, all worlds in which Donald's transaction cannot commit are eliminated. This is illustrated in Figure 2.2; we assume that there is only one available flight to LA, number 123, for simplicity. Now, suppose that Minnie submits a transaction of her own, requesting to sit next to Mickey. The new set of possible worlds is as shown in the final panel in Figure 2.2.

Suppose Mickey eventually needs to check-in and actually needs to know – i.e., read – his seat. Our goal is to make the existence of possible worlds invisible to the transaction. However, the read may have a different value depending on the possible world that it occurs in. Therefore, the system is forced to choose one possible value for the read and "collapse" the uncertainty as required so that this is the correct value read. All possible worlds consistent with the read are retained, and all others are discarded. It may so happen that some of his

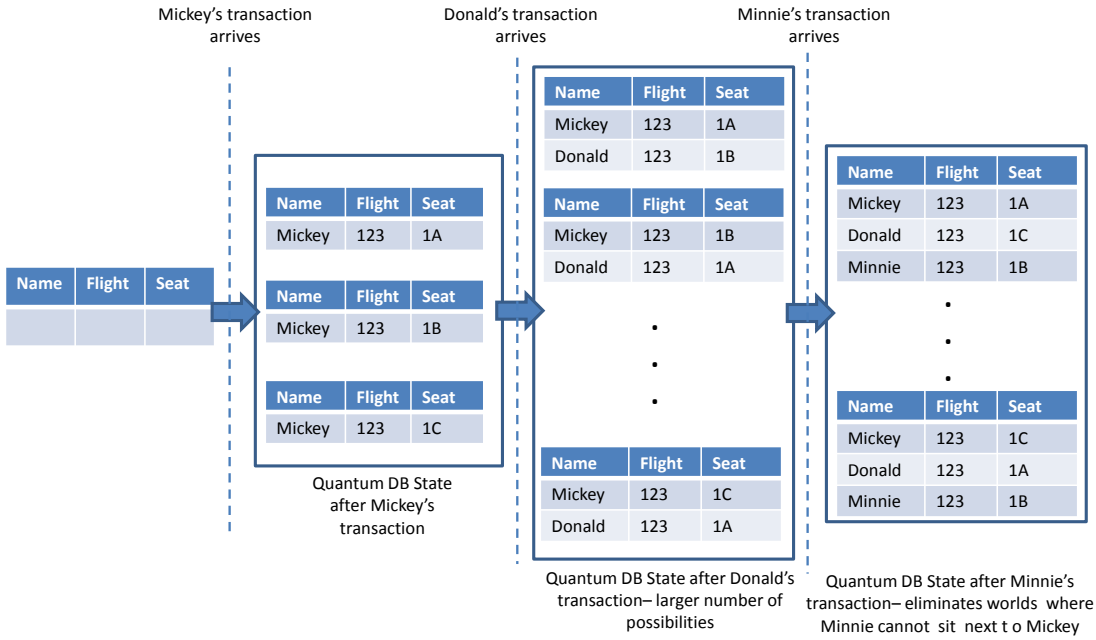

Figure 2.2: Evolution of an extensional quantum database

optional conditions cannot be satisfied in any world, in which case the world in which the maximum number of conditions are satisfied is preserved.

Of course, the number of possible worlds in the above setup would grow at an exponential rate as new transactions are processed. Therefore, an extensional representation of these worlds as described above is not practical. We can, however, represent the possible worlds *intensionally* in a concise way, and this is the idea behind our true quantum database abstraction.

At a very high level, a quantum database consists of an extensionally specified portion, which is the same in all possible worlds, and an ordered sequence of pending transactions – more precisely, committed transactions whose value assignments are still pending. To avoid cascading rollbacks and enforce serializability, it is necessary to guarantee two things – first, all the pending value assignments can be made successfully, and second, any reads on the database is-

sued by transactions serialized after a pending transaction are handled correctly. Both of these requirements are challenging to enforce. The former requires ensuring the existence of a successful grounding for the body of each pending transaction, when these transactions are executed in a sequence. The latter requires defining what dependence between transactions really means in the case of deferred value assignments, and accurately identifying pending transactions that may affect the result of a read.

More formally, we define a quantum database as follows.

Definition 2.2.1 (Quantum Database) *Let D be a completely extensional initial database. Also, let $T_0, \ldots, T_N$ be a sequence of N resource transactions. A quantum database, denoted as $\widehat{D}$, represents the set of all possible database states obtained from D by applying the operations in U_0 through U_N under consistent groundings. A grounding for transaction U_i is consistent if it corresponds to a valid grounding of B_i on the database obtained by applying U_0 through U_{i-1} to D.*

The above definition does not assume the existence of consistent groundings for the bodies. In the absence of a consistent set of groundings, the quantum database $\widehat{D}$ corresponds to $\emptyset$; during normal execution, the goal is to avoid reaching such a state. This is done by disallowing changes to both the extensional portion D and the intensional portion (the set of pending transactions) as necessary. Most concretely, if adding a new transaction to the end of the sequence would cause the set of possible worlds to become empty, the transaction is disallowed.

The above intensional representation can be modified slightly to allow semantically serializable schedules as defined in Section 2.1. This essentially in-

volves allowing changes to the ordering of the sequence of resource transactions as long as all constraints in the definition above can be maintained.

2.2.2 Maintaining a Quantum Database

We now describe how a quantum database is maintained and transformed in the presence of system operations, while satisfying the goal of retaining a nonempty set of possible worlds at all times. The operations that affect a quantum database state are reads, writes, the execution of new resource transactions, and explicit grounding (value assignment) that affects one or more pending transactions; this latter type of operation may be required for various reasons which will be explained below.

Composing resource transactions

As new resource transactions arrive and are processed, the system constructs a single logical formula whose satisfiability corresponds to the existence of a consistent set of groundings for all the pending transactions. Here, we explain at a high level how this formula is constructed. The intuition is that a sequence of resource transactions can be composed into a single resource transaction with a body that is more elaborate than the conjunction of the respective bodies. If the body of the new resource transaction is satisfiable on D, all the transactions are guaranteed successful execution (i.e., the existence of a successful value assignment). If a new resource transaction cannot be admitted in a way that preserves satisfiability, the transaction is rejected.

We begin with a key assumption about the database D. We assume that any relation R in D that appears in the `FOLLOWED BY` clause of a resource transaction has a key, i.e., satisfies set semantics. This property holds naturally in most cases; if it does not, it can be enforced either by normalization or by introducing dummy identifier columns.

We introduce a few definitions that are necessary for our presentation. Given a set of relational atoms containing variables and a database D, a *substitution* is mapping from variables to variables or data values from D. The most general unifier of two relational atoms b_1 and b_2 is defined as follows.

Definition 2.2.2 (Most General Unifier) *Let b_1, b_2 be two atoms. A unifier for b_1 and b_2 is a substitution θ such that $\theta(b_1) = \theta(b_2)$. A most general unifier (mgu) for b_1 and b_2 is a unifier θ for b_1, b_2 such that for each unifier ν of b_1 and b_2, there exists a substitution ν' for which $\nu = \nu' \circ \theta$.*

Based on the definition of most general unifier, we now define a unification predicate.

Definition 2.2.3 (Unification Predicate) *Let b_1, b_2 be two atoms. The unification predicate for b_1 and b_2, denoted φ, is a conjunction of equality constraints, where each equality constraint corresponds to a variable substitution in the most general unifier θ of b_1 and b_2.*

For example, consider the atoms $\mathtt{R}(1, v_1, v_2)$ and $\mathtt{R}(v_3, 2, v_4)$. Their most general unifier is the substitution $\{\{v_1/2\}, \{v_2/v_4\}, \{v_3/1\}\}$. Correspondingly, $\varphi = (v_1 = 2) \wedge (v_2 = v_4) \wedge (v_3 = 1)$ is their unification predicate. In the absence of a most

general unifier, the unification predicate is trivially *false*. If the most general unifier is empty (i.e., there are no variables in either atom), the predicate is trivially *true*.

We now explain how to compose a set of resource transactions into a single resource transaction. The Lemma below shows how this is done in the simple case of two resource transactions each containing a single body atom and a single atom in the update portion.

Lemma 2.2.4 *Let T_1 and T_2 be two resource transactions as given below, where each U_i and B_i is a single atom:*

$$(T_1) \;\; U_1 :-_1 B_1$$

$$(T_2) \;\; U_2 :-_1 B_2$$

A sequential execution of T_1 and T_2 on any database D is equivalent to the execution of the transaction T_{12}, given by

$$U_1, U_2 :-_1 B$$

$$\text{where } B = \begin{cases} B_1 \wedge (B_2 \vee \varphi(B_2, U_1)) & \text{if } U_1 \text{ is an insert} \\ B_1 \wedge B_2 \wedge \neg\varphi(B_2, U_1) & \text{if } U_1 \text{ is a delete} \end{cases}$$

Proof 2.2.5 *Let Var_1 and Var_2 denote the set of variables in U_1 and U_2, respectively. We assume T_1 and T_2 have no shared variables, i.e. $Var_1 \cap Var_2 = \emptyset$. Under the range restriction requirement for resource transactions, Var_1 and Var_2 are subsets of the variables in B_1 and B_2, respectively. In the proof below, we use the standard notion of valuation – a valuation is a map from variables to values in the database.*

The proof is by cases depending on whether U_1 is an insert or a delete.

Case I : U_1 *is a delete*

Consider the database state transitions $D \xrightarrow{T_1} D' \xrightarrow{T_2} D''$*, where the superscript above each arrow denotes the transaction operating on the database during the transition. Let* ν_1 *and* ν_2 *denote the valuations for* Var_1 *and* Var_2 *that lead to the database states* D' *and* D''*, respectively. We show that executing* T_{12} *on* D *under the same valuations for the respective variables leads to the same* D''*.*

Using our assumption that each relation in D *has a key, we claim that* $\nu_1 U_1 \neq \nu_2 B_2$*, that is, the relational atom deleted by* U_1 *is not the same as the relational atom that* B_2 *grounds on. Suppose this claim is false: then it must be that* $\nu_2 B_2 \in D$ *as the only difference between* D *and* D' *is the deletion of the tuple* $\nu_1 U_1$*. However, this means* D *contained two instances of the same tuple (equal to both* $\nu_1 U_1$ *and* $\nu_2 B_2$*), which is impossible by our assumption that each relation has a key.*

Establishing that $\nu_1 U_1 \neq \nu_2 B_2$ *tells us two things. First, by definition of* φ*,* $\nu_1 \nu_2 \varphi(U_1, B_2) = false$*. This implies that* $B = B_1 \wedge B_2$*. Second, we know that* $\nu_2 B_2 \in D$*. Therefore,* $\nu_1 \cup \nu_2$ *is a valid valuation for* B*, and under this valuation the execution of* T_{12} *on* D *leads to* D''*.*

Now let us consider the other direction, i.e. suppose $D \xrightarrow{T_{12}} D''$ *under valuation* ν *over* $Var_1 \cup Var_2$*. Let* ν_1 *and* ν_2 *be the projections of* ν *on* Var_1 *and* Var_2*. This implies that both* $\nu_1 B_1$ *and* $\nu_2 B_2$ *are in* D*. As* $\neg \nu_1 \nu_2 \varphi(U_1, B_2)$ *is true,* $\nu_1 \nu_2 \varphi(U_1, B_2)$ *must be false. This, by definition of* $\varphi(U_1, B_2)$*, implies* $\nu_1 U_1 \neq \nu_2 B_2$*. It follows that* $\nu_1 B_1$ *is a valid valuation for* B_1 *on* D*, and* $\nu_2 B_2$ *is a valid valuation for* B_2 *on* D'*, where* $D \xrightarrow{\nu_1 U_1} D'$ *and* $D' \xrightarrow{\nu_2 U_2} D''$*.*

Case II : U_1 *is an insert*

The reasoning here is very similar to the first case, so the proof is omitted.

Basically, the body of T_{12} reflects the fact that there needs to be a grounding for B_1 on the original database and a grounding for B_2 on the database modified by U_1; if this modification was an insert, this opens up the possibility that B_2 could ground on the inserted tuple. If U_1 was a delete, this means B_2 cannot ground on the tuple deleted by U_1. This idea extends to more general updates involving sets of inserts and deletes, and conjunctive queries with multiple body atoms.

Theorem 2.2.6 (Composition) *Let T_1 and T_2 be two resource transactions as given below:*

$$(T_1) \quad U_1 :- B_1$$

$$(T_2) \quad U_2 :- B_2$$

A sequential execution of T_1 and T_2 on any database D is equivalent to the execution of the transaction T_{12}, given by

$$(T_{12}) \quad U_1, U_2 :- \quad B$$

$$\text{where } B = B_1 \wedge \bigwedge_{i,j} (b_2^i \wedge \neg\varphi(b_2^i, d_1^j)) \wedge \bigwedge_{i,j} (b_2^i \vee \varphi(b_2^i, i_1^j))$$

and $b_2^i \in B_2$ and d_1^j, i_1^j are the deletes and inserts in U_1 respectively.

Proof 2.2.7 *This can be shown with a generalization of the argument used to prove Lemma 2.2.4.*

An example of the composition of three resource transactions is shown in Figure 2.3. In the first transaction, Mickey cancels a reservation for a seat on flight number `1`. In the second, Donald books a seat on a flight, where both the

$$
\begin{array}{ll}
(T_1) & -\mathtt{B}(\mathtt{M}, 1, s_1), +\mathtt{A}(1, s_1) :\!-_1 \ \mathtt{B}(\mathtt{M}, 1, s_1) \\
(T_2) & -\mathtt{A}(f_2, s_2), +\mathtt{B}(\mathtt{D}, f_2, s_2) :\!-_1 \ \mathtt{A}(f_2, s_2) \\
(T_3) & -\mathtt{A}(2, s_3), +\mathtt{B}(\mathtt{G}, 2, s_3) :\!-_1 \ \mathtt{A}(2, s_3)
\end{array}
$$

(a)

$$
(T_{12}) \ \ U_1, U_2 :\!-_1 \mathtt{B}(\mathtt{M}, 1, s_1) \wedge \{\mathtt{A}(f_2, s_2) \vee \{(f_2 = 1) \wedge (s_1 = s_2)\}\}
$$

$$
\begin{array}{ll}
 & \mathtt{B}(\mathtt{M}, 1, s_1) \wedge \\
(T_{123}) \ U_1, U_2, U_3 :\!-_1 & \{\mathtt{A}(f_2, s_2) \vee \{(f_2 = 1) \wedge (s_1 = s_2)\}\} \wedge \\
 & \mathtt{A}(2, s_3) \wedge \neg\{(f_2 = 2) \wedge (s_3 = s_2)\}
\end{array}
$$

(b)

Figure 2.3: (a) Three resource transactions (b) The composition of the first two and all three transactions from (a) above; each U_i denotes the update portion of transaction i.

seat and the flight are unconstrained. In the third, Goofy books a seat on flight 2. Figure 2.3 (b) shows the composition of the first two transactions, and then the composition of all three. The satisfiability of the body of T_{123} guarantees that there is a possible value assignment corresponding to the ordered execution of the three original transactions.

Reads and Writes

We now explain how a quantum database manages the set of possible worlds it represents when handling reads and writes.

Reads: What should happen when a quantum database representing a set of possible worlds is read? For example, what happens if Mickey submits a query to find out his actual flight and seat number? There are several options. One is to expose the uncertainty to the user by returning all possible values for the read. The second is to pick a single value and return it, which can be done under

two different semantics. First, we can pick a single possible value and return it without nonetheless fixing it in the database, so that Mickey would see a particular seat number but have no guarantees that this number will remain fixed. Second, we can pick a single value at query answering time and fix it at that time, essentially "collapsing" part of the quantum state to a concrete, extensionally specified state. These three options represent different points in a trade-off between providing strong read consistency guarantees and maintaining a large set of possible worlds to allow for better future resource allocation.

The quantum databases we present in this dissertation use the third option mentioned above; this approach completely hides the uncertainty and allows the programmer to assume that he or she is working with a standard database that provides the expected read repeatability guarantees. In practice, we expect reads to be infrequent until the time of resource "consumption" (such as checking in for a flight), at which time of course the assignments must be fixed anyway. We note in particular that the programmer is notified by the system when his or her resource transaction commits; this notification is a guarantee that a suitable resource exists and will still exist when it is actually needed in the future. Therefore, there is no need for the programmer to issue an immediate read to "check" whether a suitable resource for the transaction exists in the system – the fact that the transaction committed already provides the desired confirmation.

In certain application-specific settings, handling reads in a way that exposes uncertainty and/or loosens consistency guarantees may be an acceptable solution. For example, consider the calendar management scenario we introduced in Section 1.1. We could use a quantum database to delay finalizing employee

meeting schedules until absolutely necessary. However, it might be useful for employees to know *some* information about their schedules a day or two in advance: for example, they may want to know whom they are meeting so that they can prepare appropriately, without needing to know exactly when each meeting occurs. This would suggest a more complex read model where reading an employee's schedule for that day removes some of the uncertainty – regarding whom he or she is meeting – but retains the uncertainty regarding the specific time of each meeting. Developing quantum databases that use such alternate approaches for read handling is future work.

The approach we take in this dissertation, i.e., fixing a particular value assignment at read time, obviously reduces the opportunities available for future optimization. Two important challenges arise: what values must be fixed when handling a particular read, and how should they be fixed? Both of these are nontrivial, and it is desirable to resolve them intelligently.

Identifying the values that must be fixed to handle a read can be done at different levels of precision. At the highest level of generality, this question is related to the problem of computing information disclosure through views, which is Π_2^p-complete [75]. However, a simple practical solution is to use a conservative criterion based on unifiability. If a relational atom in our incoming read query unifies with a pending update U_i from a transaction T_i, the values involved in that transaction are fixed.

Certain reads necessitate more grounding than others. For example, a read requesting the full contents of the `Booking` table will cause many more groundings than a read asking only for Mickey's seat number. The programmer should be aware that such general reads have a potential negative impact on optimal re-

source allocation and should strive to avoid them. It is also possible to imagine solutions where the programmer is provided more explicit feedback before issuing a read on the potential "consequences" of that read on the possible worlds. This might be helpful to the programmer as he or she determines which reads to issue; on the other hand it violates the pure quantum database model where uncertainty is totally hidden from the programmer. Investigating the details and usefulness of such a solution is ongoing work.

Once the system decides which values need to be fixed due to a read, the system may still need to make a choice if more than one satisfying assignment is available for the transactions in question. Which specific seat among all those that are open should Mickey receive? Generally, it is desirable to fix values in such a way as to maximize the remaining number of possible worlds; more sophisticated application-specific heuristics may also be appropriate.

Writes: Writes are significant in a quantum database because they may cause the formula associated with the pending transactions to become unsatisfiable. Thus, all writes to the database which unify with the bodies of the pending transactions need to pass through a check and are rejected if the check fails. This is analogous to the check performed when processing a new resource transaction, with the difference that a blind write changes the database over which the formula must remain satisfiable rather than the formula itself.

Grounding

The uncertainty in the quantum database may need to be resolved at certain points, due to a read or due to application specific requirements. Fixing some

concrete value assignments in the database may require actual execution of some of the committed resource transactions that were previously pending value assignment.

When a particular value assignment must be fixed and a particular pending transaction actually carried out, the naïve approach would be to also ground and apply all transactions that arrived in the system earlier. For example, consider a quantum database $\widehat{D}$ with pending transactions $T_0, \ldots, T_N$ over an extensional database D. Suppose that transaction T_i requires grounding. The naïve approach is to apply transactions $T_0, \ldots, T_i$ in turn on the database D to get a new database D', by grounding each of them and carrying out the appropriate update. The definition of a quantum database guarantees that suitable groundings exist for each of $T_0, \ldots, T_i$ as long as the set of possible worlds is nonempty. This procedure yields a quantum database with extensional state D' and the pending updates of $T_{i+1}, \ldots, T_N$. Such an approach provides strong serializability of the transactions in arrival order; however, it may overconstrain some assignments prematurely and reduce the window of opportunity for future optimization.

An alternative is to strive for *semantic serializability* (as introduced in Section 2.1); this approach avoids grounding transactions unless strictly necessary. The invariant associated with the quantum database $\widehat{D}$ is, in general, order dependent. While this invariant guarantees successful execution for the order in which the transactions arrived, there can be other orderings of the pending transactions which have the same desired effect. Of course, reordering the transactions affects the formula introduced in Section 2.2.2 whose satisfiability guarantees successful execution of the remaining pending transactions. This means we cannot reorder pending transactions arbitrarily, but must check for satisfia-

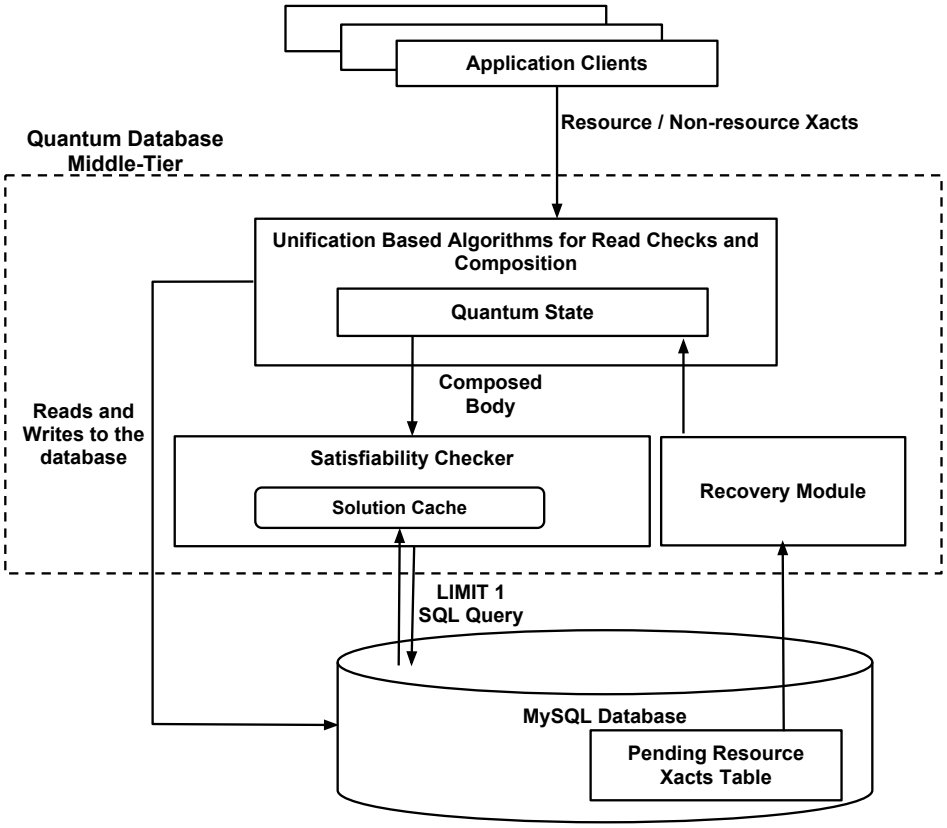

Figure 2.4: Quantum Database System Architecture

bility first. That is, we must maintain the invariant that there exists some ordering of the remaining pending transactions under which the resulting formula is still satisfiable. Checking for all possible reorderings would be computationally intractable as there are exponentially many of them. A practical strategy is to check only the ordering where the transaction under consideration is moved to the front of the current ordering. In most cases, we expect the underlying satisfiability problem to be very under-constrained (i.e. many available resources and few pending transactions) so this strategy should yield good results.

2.3 Quantum Database Prototype

Our quantum database prototype is implemented as a middle-tier service over a MySQL database. From the developer's perspective, the API is almost identical to the API provided by any standard database. It allows the developer to submit queries and updates to the database; the major new feature is support for resource transactions. Our current implementation does not accept and parse resource transactions in their SQL format, but only in the intermediate Datalog-like representation.

Figure 2.4 shows the architecture of our quantum database prototype. The constraint satisfiability checking required to maintain the quantum database invariant is performed using database `LIMIT 1` queries; alternate possible solutions are discussed in Section 2.5. We explain some of the important architectural features of our prototype below.

Quantum State: The prototype keeps an in-memory representation of the intensional portion of the quantum database state. This in-memory state is maintained as a set of composed transaction bodies, where each composed body is a single formula that looks like the one in Theorem 2.2.6. Some resource transactions are totally independent of each other, i.e., there is no unification possible between them – this is true for example of transactions that book seats on different and explicitly specified flights. The system partitions the resource transactions accordingly into independent sets and maintains a separate composed transaction body for each set. This partitioning obviously helps keep all the required computations efficient as the quantum state evolves. The partitioning is not fixed as new transactions arrive, however. For example, we may

have a scenario involving two sets of transactions, one containing only requests for window seats and the other only for aisle seats. These sets are independent; however, if a new transaction arrives requesting either a window or an aisle seat, then this new transaction as well as the two original sets must all be merged and composed together.

Solution Cache: The prototype maintains an in-memory cache of possible solutions (i.e., value assignments) to the composed transaction bodies. Recall that a quantum database must maintain the invariant that there exists at least one grounding for each of the composed bodies. When a new resource transaction arrives in the system, we check whether an existing solution in the cache can be extended to accommodate the new transaction. If this is not possible, then we generate a `LIMIT 1` SQL query corresponding to the body of the new composed transaction and send it to the database. If this query has an answer, the solution cache is updated appropriately and the transaction commits. If not, then the new resource transaction is aborted. Maintaining a solution cache allows us to amortize the cost of checking satisfiability of composed bodies across a set of transactions. However, in the worst case, the search for a solution must be restarted from scratch each time. A strategy to avoid such recomputation is to increase the number of solutions maintained in the cache. Such additional solutions can be computed by a background process in order to keep the per-transaction latency low. Our current prototype does not implement this strategy, but instead maintains a single solution in the cache for every composed transaction.

Since the solution cache always contains at least one valid grounding for all the composed transactions, read queries which induce grounding of pending

transactions can be answered without much overhead. If a read requires fixing some values in the database, the system can use appropriate values from the solution cache to apply the updates of the affected resource transaction(s). Once the required values are fixed in the database, the read query is processed normally.

Recovery: Since the execution of resource transactions is deferred post-commit, we need to maintain additional information about these transactions to ensure durability. We do this by utilizing the recovery mechanisms of the underlying database. Each pending resource transaction is serialized and inserted into a special database table called the *pending transactions table*. This insertion happens after the satisfiability check and before the transaction commits. During recovery, a quantum database module restores the in-memory quantum state to what it was before the crash based on the pending transactions table. When a pending resource transaction is grounded and executed, it is removed from the pending transaction table.

Our prototype is implemented as a Java application built over MySQL (version 5.5.28) using the InnoDB engine. The maximum number of relations that can be referenced in a single MySQL join operation is limited to 61; this means our system can only handle up to 61 atoms in each composed transaction body. This limitation is not fundamental to the problem itself, but arises out of the maximum number of joins supported by MySQL. The semantics of quantum databases allows the reduction of uncertainty through grounding at any time; therefore, we keep the size of the composed bodies small by forcibly grounding and executing some pending resource transactions as needed. Concretely, we ground transactions to keep the maximum number of pending transactions in

each partition below a parameter k; when grounding, we start with the oldest transactions based on their arrival time in the system.

2.4 Experiments

In this section, we show the results of an experimental evaluation of quantum databases in a realistic application setting. The aim of our evaluation is primarily twofold. One, to measure the overhead of quantum databases over traditional databases, and second, to quantify the improvement in allocation of resources due to deferred execution through quantum databases.

2.4.1 Application Scenario

Our experiments are set in the travel application scenario used throughout the chapter, enhanced with the presence of user-defined coordination constraints that are expressed as entangled queries [46]. Entangled queries allow users to build powerful applications where the heart of the coordination – the choice of data values – is performed at the same declarative level as the data access. Entangled transactions [47] are transactions which include entangled queries. The algorithms presented in Section 2.2.2 can be modified easily to include functionality related to entanglement and turn quantum databases into a platform for executing entangled resource transactions.

For example, the transaction in Figure 2.1 can be read and executed as an entangled resource transaction, in the following way. If Mickey submits this transaction before Goofy has arrived in the system, the system can maintain Mickey's

request to sit next to Goofy as a "forward constraint," to be satisfied if possible should Goofy arrive in the system later. If Goofy does arrive, the system will try to give him and Mickey adjacent seats. Of course, if Goofy never arrives, coordination is impossible, which is why Mickey's coordination constraint needs to remain `OPTIONAL`.

Entangled resource transactions are different from the (pure) entangled transactions [47] where coordination was required for successful execution. The execution model for entangled transactions does not allow an entangled transaction to commit until its partner(s) is also in the system; this means that entangled transactions must be executed in batches. Our new quantum database model allows entangled resource transactions to execute and commit individually. The benefit of the current approach is that while Mickey is no longer guaranteed to sit next to Goofy, he is now guaranteed to have a seat for himself regardless of when and whether Goofy might submit his transaction.

Since the primary motivation for deferring the execution of an entangled resource transaction is to allow coordination with a yet-to-arrive partner transaction, an entangled resource transaction waiting for its partner is finally executed as soon as its partner arrives and no longer remains in a quantum state. That is, when both coordinating users' transactions are in the system, their respective seat assignments are fixed. This situation is an example where the application logic decides how long a resource transaction should be kept in a quantum state.

Order of Arrival	Characteristic	Max. Number of Pending Xacts
`Alternate`	T_i entangles with T_{i+1}	1
`Random`	T_i entangles with T_j for some $i, j < N$	$\lceil N/2 \rceil$
`In Order`	T_i entangles with $T_{i+N/2}$	$\lceil N/2 \rceil$
`Reverse Order`	T_i entangles with T_{N-i}	$\lceil N/2 \rceil$

Table 2.1: Four different transaction arrival orders and the maximum number of pending transactions in the quantum database assuming a transaction remains pending until its partner arrives.

2.4.2 Experimental Setup

We created a workload of simulated entangled resource transactions to model the output of the front-end social travel application as described above. Our workload simulates users desiring to coordinate with their friends on flights and to sit in adjacent seats. We compare this workload against a workload of non-entangled transactions issued by an "intelligent social" (`IS`) user. Such a user first issues a query to check whether his/her friend has an existing reservation. If so, he books the adjacent seat, and if not he books a seat with a free adjacent seat. The `IS` workload simulates the kind of coordination that is achievable without using a quantum database.

The overhead of quantum databases depends on the complexity of checking the invariants maintained by the quantum database. Each invariant corresponds to the body of a composed transaction, and hence the complexity of checking the invariant depends on the number of pending transactions which are composed together. As described in Section 2.4.1, an entangled resource transaction is kept pending only until the arrival of its partner. Therefore, the order of arrival of the transactions w.r.t. their partners determines the complexity of the invariants. Table 2.1 shows four possible orders of arrival of transactions.

In the `Alternate` arrival order, each user transaction is followed immediately by his or her partner's transaction. This only leaves a maximum of 1 pending transaction in the quantum database. The second possible order of arrival is `Random`, which orders transactions randomly and is expected to be the most realistic. While the maximum number of pending transactions for the `Random` order is $N/2$, it is expected to be lower on average. Finally, the orders of arrival which lead to over half the total number of transactions to be pending are denoted as `In Order` and `Reverse Order`. In `In Order` order of arrival, half the users submit their transactions followed by their respective partners in the same order. In `Reverse Order`, the second half of the users submit transactions in the reverse order, i.e., the first user sits entangles with the last user, the second user entangles with the second to last user and so on. While the maximum number of pending transactions for both `In Order` and `Reverse Order` are the same, for `Reverse Order` the period for which a transaction is kept pending varies from 1 to N, as opposed to a constant $N/2$ for `In Order`.

We artificially generate a database of flights over which the reservation requests are issued. Each flight in our database is represented as a set of seats arranged in rows of three. Each row has four possible adjacent pairs, only two of which can be booked simultaneously. The number of rows per flight and the number of flights in the database are changed across experiments. Appropriate indices are defined for each relation in the database.

In all our workloads, all coordination partners arrive in the system at some point so full coordination is theoretically achievable. A key metric for measuring the benefit of quantum databases is the percentage of maximum possible coordination which is actually achieved. For example, for a single flight in our

database with ten rows (10×3 seats), a maximum of twenty coordination requests for adjacent seats can be accommodated, and therefore the benefit of using quantum database is measured as the fraction of users (of the maximum twenty) who actually are able to coordinate. The expectation is that quantum databases should allow us to significantly increase this percentage over what is possible with an intelligent social strategy.

We ran all experiments on a 2.13GHz Intel(R) Xeon(R) E5606 with 48 GB of RAM. The MySQL query optimizer by default performs an exhaustive search over all possible query plans, and this number grows exponentially with the number of tables referenced in a join query [1]. Quantum database queries typically involve a high number of joins, and therefore for default values the database is observed to spend a disproportionate amount of time in query optimization as compared to query execution. For all our experiments, we set the value of the parameter `optimizer_search_depth` to 3 to reduce the amount of time spent in query optimization without significant change in query execution time. The reported values are averages over 5 runs. Executable `.jar` files and instructions for replicating our experiments can be found on our project website [2].

2.4.3 Results

Order of arrival: The first experiment measures the overhead of using a quantum database for the four different orders of arrival in Table 2.1. We set the parameter for k to its maximum value of 61 for this experiment. We start with a database containing a single flight with 102 seats (34 rows of 3 seats each), and

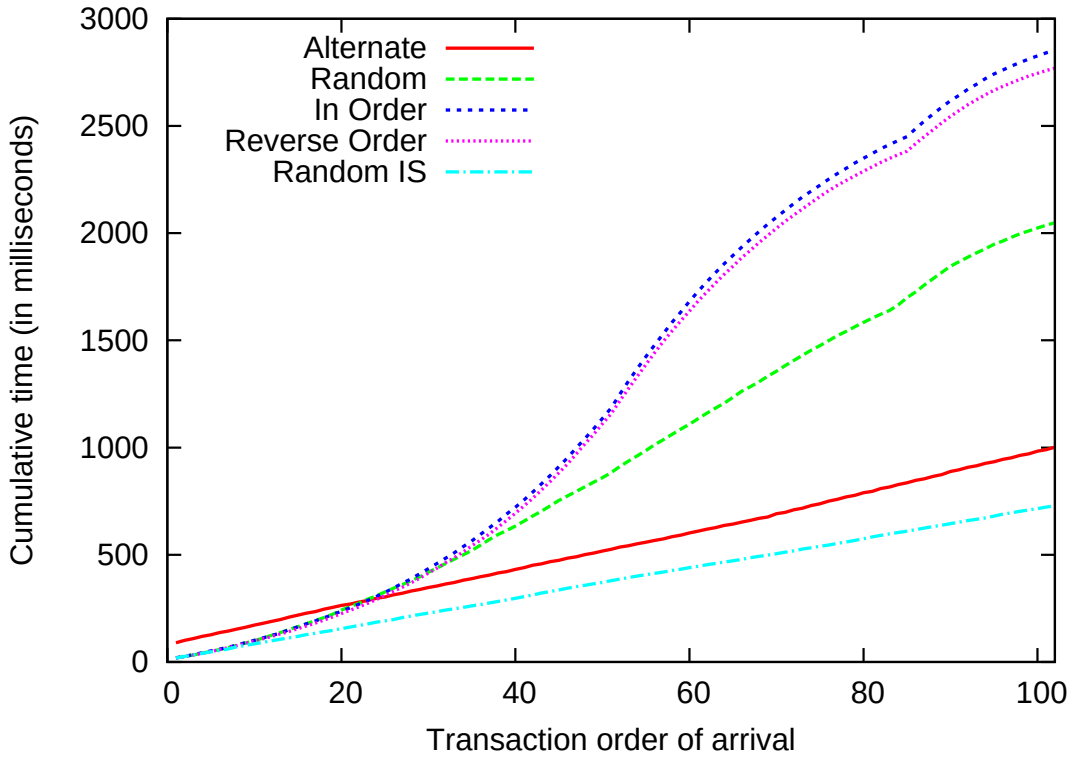

Figure 2.5: Cumulative time of transaction execution for different orders of arrival of transactions.

issue a sequence of 102 transactions according to each order. Our choice of the value of k ensures that the partner for a transaction arrives within the window provided by k.

Our results are shown in Figure 2.5 and Figure 2.6. We ran the experiment with entangled and intelligent social workloads for each of the four transaction arrival orders described above. We found that the performance of the system on the intelligent social workload does not depend on arrival order, so we only show results for the four different arrival orders of the entangled workload and for the intelligent social workload in the `Random` arrival order. Figure 2.5 shows the cumulative execution time for each workload. Under the `Alternate` arrival order, the overhead of our system as compared to intelligent social is neg-

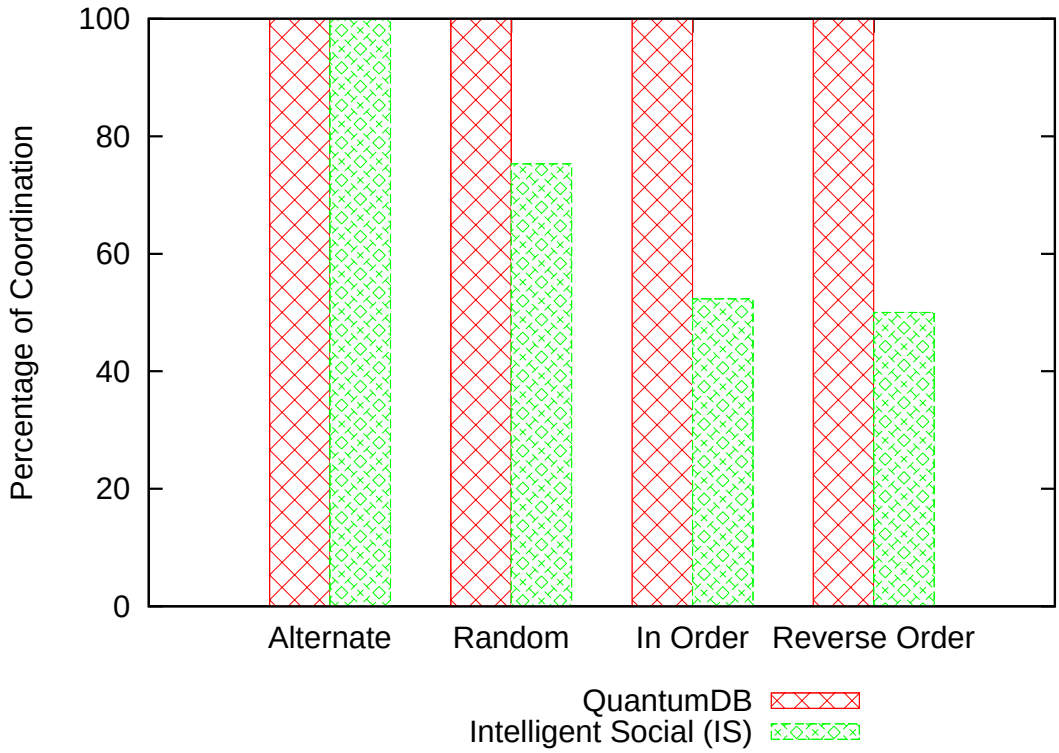

Figure 2.6: Percentage of coordination for different orders of arrival of transactions.

ligible. This is expected, as only a maximum of one transaction is kept pending by the quantum database. However, for both `In Order` and `Reverse Order` arrival orders, our system is substantially slower than the intelligent social approach. The steep slopes for both these arrival orders in Figure 2.5 are caused by the increasingly larger bodies of the composed transactions. As the partner queries start to arrive in the second half of the respective workloads, the slope reduces as the number of pending transactions decreases. The `Random` arrival order, which we expect to be by far the most realistic, shows a small overhead over intelligent social, on the order of a few milliseconds per transaction. We believe this is acceptable in practice, particularly given the significant increase in successful coordination that is gained through using a quantum database.

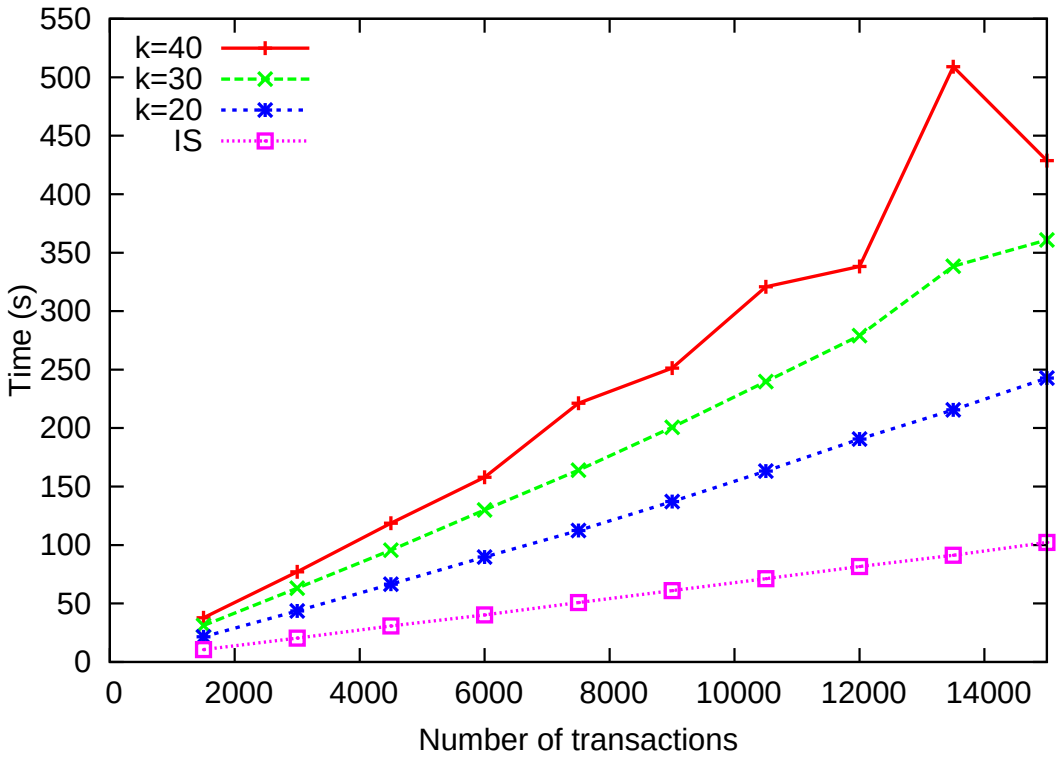

Figure 2.7: Scalability

Figure 2.6 shows the percentage of the total possible coordination that the system actually achieves for each arrival order. The quantum database achieves the maximum possible coordination for all of the four workloads; for all arrival orders except `Alternate`, this is a substantial improvement over what is possible with the intelligent social approach.

Scalability: We test the scalability of our system as the number of flights in the database is increased from 10 to 100. Each flight in the database has 150 seats (50 rows of 3 seats each). We initialize the system in a state where all flights are fully available, and issue as many transactions as there are available seats in `Random` order. Upon completion of all transactions each user has a seat and all available seats are booked. Instead of maintaining a single large invariant for all flights, the system correctly identifies the independence of queries between

different flights, and it maintains a relatively smaller set of invariants, one for each flight. Such partitioning allows the system to scale as the number of flights is increased.

Figure 2.7 shows the total time taken for all transactions to complete and Table 2.2 shows the percentage of successful coordination among the transactions. The percentage of coordination is dependent on the maximum number of pending transactions per flight and as expected remains constant with respect to the number of transactions. The average values of the percentage of coordination for different values of k is shown in Table 2.2. From Figure 2.7 it can be seen that with smaller values of k, execution is faster since the body of the composed transaction we must maintain and check satisfiability on has a smaller number of joins. However, the percentage of successful coordination is lower as the system grounds transactions pre-emptively reducing the chances of successful coordination. Even for small values of k, we see a factor of 2 improvement in coordination percentage over the `IS` strategy.

We investigated the unexpected increase in Figure 2.7 for $k = 40$ at $13,500$ transactions. This increase occurs due to an unexpectedly large amount of time spent in finding the solution to the composed body of a few transactions. Digging deeper, we saw that this is an artifact of the MySQL query optimizer which chooses a bad query plan for several quantum database queries. We observed that such queries occurred rarely and only for certain random orderings. We modified the value of `optimizer_search_depth` to find an alternate plan, and with an alternate plan the problem queries could be answered about two orders of magnitude faster. We believe better query plans would eliminate this erratic behavior. An alternate ad-hoc approach would be to ground some pend-

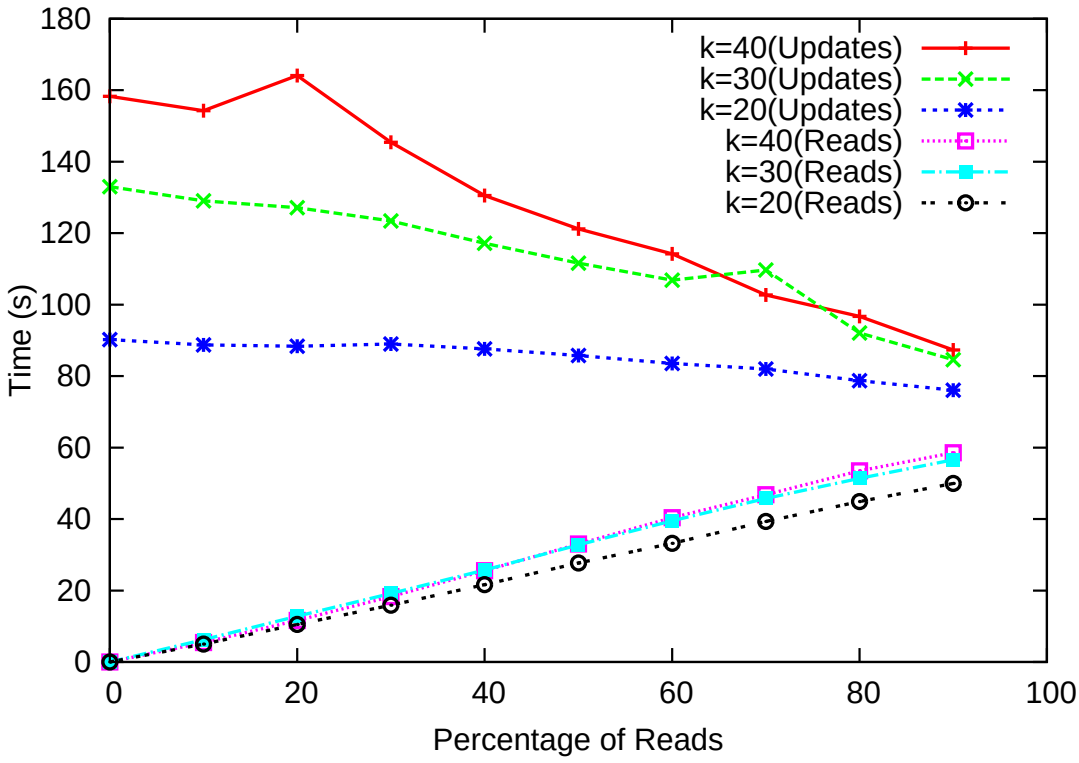

Figure 2.8: Performance under mixed workload.

Quantum DB			Intelligent Social
k=20	k=30	k=40	
45.6	86.9	99.9	20.2

Table 2.2: Average percentage of successful coordinations

ing transactions to keep the query complexity and the resulting query execution times within acceptable limits. A more fundamental approach to solving this problem is discussed in next section.

Our system scales linearly in terms of execution time and maintains a constant coordination percentage as the number of transactions increases. This linear increase is a consequence of the non-unification based partitioning strategy implemented by the quantum database.

Mixed Workload: Next, we study the behavior of our system under realistic

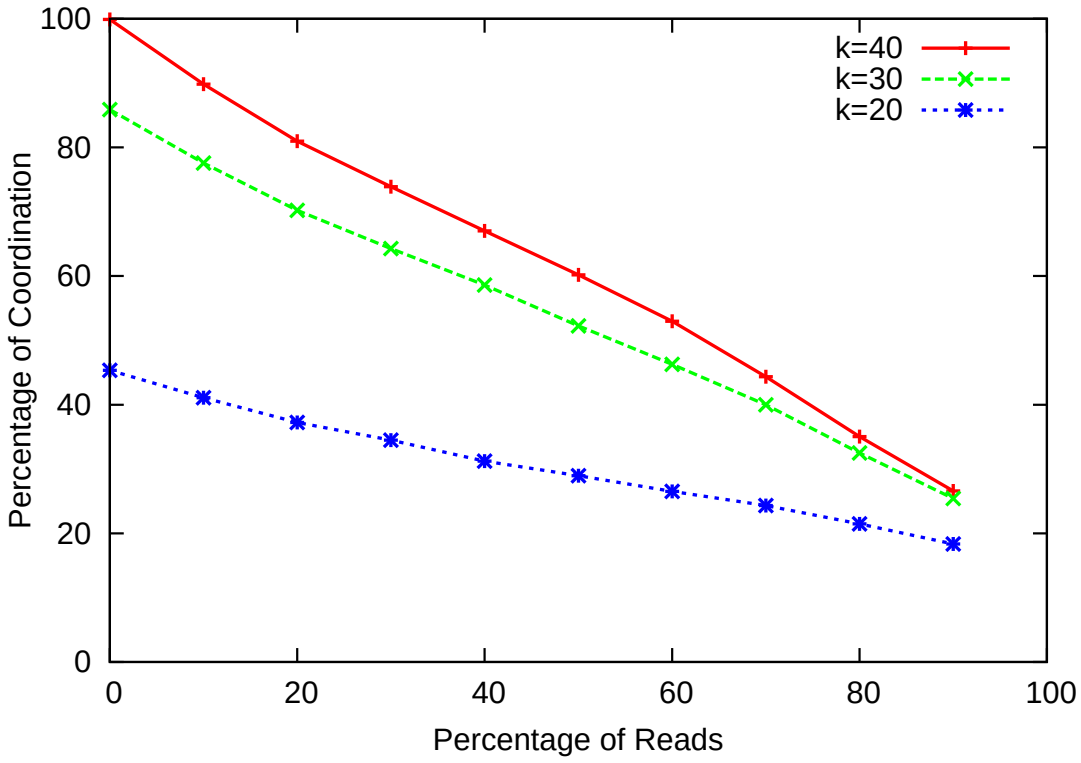

Figure 2.9: Percentage of coordination w.r.t. percentage of reads.

workloads which are a mix of resource and non-resource transactions. The non-resource transactions are read queries by users who had earlier issued a resource transaction. Unlike in normal databases, a non-resource read transaction on a quantum database can induce updates to the database by forcing grounding of pending resource transactions. Such forced grounding, however, reduces the chances of successful coordination.

We run our experiment over a random ordering of 6000 transactions. The database contains 40 flights each with 150 seats (50 rows of 3 seats each). This ensures after all transactions are executed, every user has a seat. We increase the number of read transactions in steps of 10% (600 transactions) from 0% to 90% (5400 transactions), and use values of k ranging from 20 to 40. As shown Figure 2.9, the percentage of successful coordination decreases linearly with the

percentage of reads. This is expected as under a high read workload, the system needs to pre-emptively fix many transactions and thereby prevents coordination. This is corroborated by Figure 2.8 which shows an increasing amount of time spent on the reads and a decreasing amount of time spent on executing the resource transactions. The decrease in the time spent on resource transactions is because of preemptive grounding of pending transactions on being read. This leads to fewer pending transactions in the system, and correspondingly simpler satisfiability checks.

The anomalous behavior in Figure 2.8 at 20% reads for $k = 40$ and 70% reads for $k = 30$ can be attributed to the choice of bad query plans by query optimizer as observed and explained in the previous experiment.

2.5 Discussion

Making quantum databases a practical platform for developing applications involving resource allocation raises exciting research challenges. In this section, we highlight some of these challenges and discuss potential solutions.

Efficiency of evaluation: A core aspect of maintaining a quantum database is checking and maintaining the satisfiability of the composed transaction formula. Implementing the satisfiability check in the naïve way with relational queries is suboptimal in the general case, as the resulting query has a large number of joins and is not well-suited to the capabilities of a traditional relational optimizer. We discuss two potential solutions to overcome this limitation of quantum database.

First, this problem is an instance of the Satisfiability problem, which is known to have phase transitions [95]. Most real-world resource allocation problems are under-constrained at least in the initial phase, e.g., when seats on a given flight first go on sale. As resources are allocated the problem tends to reach a critical ratio of degrees of freedom (variables) vs. constraints at which the problem is hard; both comfortably under- and over-constrained problems tend to be easy to solve [95]. By identifying the difficulty of checking satisfiability, a quantum database can switch to a more aggressive fixing phase favoring faster response times over better assignments.

Second, we believe, we can leverage state-of-the-art Satisfiability Modulo Theory (SMT) solvers [30], which are essentially SAT solvers where the interpretation of some symbols is constrained by a background theory. Identifying the appropriate background theory for quantum databases and designing an algorithm which splits the task of satisfiability checking between the database and the solver to achieve maximum efficiency requires addressing many research problems, both theoretical and systems-related.

System design: Another key issue is the integration of quantum database functionality into the technology stack used by developers. Should quantum databases be implemented within or on top of a DBMS? The latter approach may be simpler and more portable, but the former should yield better performance as well as significant systems insights as we investigate how to integrate quantum database functionality with the internals of a DBMS.

The quantum database model for resource allocation has conceptual connections with existing concurrency control algorithms already implemented in the DBMS to maintain isolation and to perform recovery. Consider isolation first:

in a crude sense, a transaction that has committed in a quantum database has a "logical lock" on an instance of the resource and the quantum database system is performing logical lock management. It is this logical locking which allows reordering of logically non-conflicting transactions as explained in Section 2.2.2. While we have presented a practically viable approach for detecting such logical lock conflicts using unification, it will be interesting to understand how the algorithms for maintaining a quantum database can be optimized by offloading parts of it to the underlying database. Conversely, the algorithms for maintaining a quantum database may provide insights for designing intention based concurrency control mechanisms. Regarding recovery, in many ways quantum databases avoid rollbacks by performing resource allocation late and dropping constraints if they cannot be satisfied. The question arises of whether this in any way relates to the work performed by the recovery manager. Even if not, it is possible that the recovery manager should be designed in a different way to better support the unique features of quantum databases.

2.6 Summary

In this chapter, we presented quantum databases, a novel abstraction for declarative resource allocation. We introduced the idea of deferred execution of transactions to improve allocation of resources in a dynamic system. To make this idea practical, we proposed unification based algorithms for efficiently maintaining the database in a partially intensional representation in the presence of queries and other transactions. We evaluated the performance of quantum databases over our prototype implementation in a realistic application scenario, and demonstrated the improvement in allocation of resources due to deferred

execution. We believe that research to address the quantum database-related research challenges which we identified will expand the power of and provide insights into multiple aspects of database engines.

CHAPTER 3

HOMEOSTASIS

If a tree falls in a forest and no one is around to hear it, does it make a sound?

— George Berkeley

Datastores today rely on distribution and replication to achieve improved performance and fault-tolerance. But correctness of many applications depends on strong consistency properties—something that can impose substantial overheads, since it requires coordinating the behavior of multiple nodes. This chapter describes a new approach to achieving strong consistency in distributed systems while minimizing communication between nodes. The key insight is to allow the state of the system to be inconsistent during execution, as long as this inconsistency is bounded and does not affect transaction correctness. In contrast to previous work, our approach uses program analysis to extract semantic information about permissible levels of inconsistency and is fully automated. We then employ a novel *homeostasis protocol* to allow sites to operate independently, without communicating, as long as any inconsistency is governed by appropriate *treaties* between the nodes. We discuss mechanisms for optimizing treaties based on workload characteristics to minimize communication, as well as a prototype implementation and experiments that demonstrate the benefits of our approach on common transactional benchmarks.

Homeostasis is the property of a system in which variables are regulated so that internal conditions remain stable and relatively constant.

3.1 Analyzing Transactions

This section introduces our program analysis techniques that capture the abstract semantics of a transaction in *symbolic tables*. We first introduce our formal model of transactions (Sections 3.1.1, 3.1.2) and symbolic tables (Section 3.1.3). Then, we present a simple yet expressive language and explain how to compute symbolic tables for that language (Sections 3.1.4, 3.1.5). In practice, the idea is that transaction code in a language such as SQL can be compiled automatically into the language from Section 3.1.4 and fed into the analysis algorithm to produce symbolic tables.

3.1.1 Databases and transactions

We model a database as a set of finite arrays of integers. Formally, let Arr be a countably infinite set of array names denoted by $a, b, c, \ldots$. A database D is a finite subset $A \subseteq Arr$, where each $a \in A$ has an associated partial function $a : \mathbb{Z} \to \mathbb{Z}$. All arrays are finite, i.e., each function a is defined only over a finite subset of $\mathbb{Z}$. Given an integer i, $a(i)$ denotes the value of function a for i, or more intuitively the ith element of array a. $a\{i \leftarrow j\}$ denotes the array a modified so that $a(i) = j$. Let $\mathcal{D}$ be the universe of all possible databases.

For $a \in Arr$, $i \in \mathbb{Z}$, if $a(i)$ is defined in D, we call $a(i)$ a *database object* in D and the pair $\langle a, i \rangle$ is the *name* of the object. All possible object names are thus drawn from the set $Arr \times \mathbb{Z}$; we denote this set as Var and call its elements *database variables*.

Some arrays in D may contain only a single integer. Such singleton arrays are

useful for use in examples. To streamline notation, we introduce *integer database objects* as syntactic sugar for singleton arrays. That is, we allow objects whose name is a singleton a rather than a pair $\langle a, i \rangle$. To make it clear we are working with such objects, we name them $x, y, z...$ rather than $a, b, c...$

A *transaction* T is an ordered sequence of atomic *operations*, where an operation is either a read, a computation, or a write. A read retrieves a value from the database and stores it into a temporary program variable, a write stores a concrete value into a specific object in the database, and a computation neither reads from nor writes into the database, but may update temporary variables. Situations where the transaction code interacts with the external world are modeled by treating input statements as reads, and output statements as writes on fresh and unique data items.

3.1.2 Read and write traces

Consider a transaction T running on a database D. The *read trace* of T on D, denoted $RT(T, D)$, is a sequence of pairs (x, v) where x is a database object – either an array element or an integer – and v is the value of x as seen by T during a specific read. The *write trace* of T on D, denoted $WT(T, D)$ is defined analogously as the sequence of values written by each of T's writes. For elements of write traces, we use the notation $x := v$ to represent more clearly that T is writing value v to object x.

We now consider the execution of T on various databases and observe similarities between some of these executions. First, there are some databases on which T reads the same values—i.e. has the same read trace. We can define the

following equivalence relation:

Definition 3.1.1 (Read-trace equivalence) *Given a transaction T, we define* read-trace equivalence *as a relation on $\mathcal{D}$. For any $D, D' \in \mathcal{D}$ we say that D and D' are read-trace equivalent for T (we write $D \equiv_T^R D'$) if and only if $RT(T, D) = RT(T, D')$*

Analogously, we can define a write-trace equivalence relation $\equiv_T^W$ using write traces instead of read traces.

Example 3.1.2 *Consider a transaction T that reads the value of x from the database and writes $y = 1$ if x is positive and $y = 0$ otherwise. The databases $\{x = 5, y = 6\}$ and $\{x = 4, y = 6\}$ are write-trace equivalent for T, but not read-trace equivalent.*

We make the following assumption, which is standard:

Assumption 3.1.3 (Determinism) *For any transaction T and any $D, D' \in \mathcal{D}$, if $RT(T, D) = RT(T, D')$ then $WT(T, D) = WT(T, D')$*

Given a transaction T, we can construct the quotient set of $\mathcal{D}$ by the relation $\equiv_W^T$, i.e. a partition of $\mathcal{D}$ into the equivalence classes of $\equiv_W^T$, where each equivalence class has a concrete associated write trace. This is denoted as Q_T; if $\mathcal{D}$ is finite it can be computed using a brute-force algorithm that runs T on every possible database. Consider the transaction in Figure 3.1a, where $\hat{x}, \hat{y}$ are temporary variables. Assume x and y are both restricted to take only values between 0 and 5. The quotient set for T_1 is shown in Figure 3.1c.

$T_1 ::= \{\hat{x} := \texttt{read}(x);$
$\quad \hat{y} := \texttt{read}(y);$
$\quad \texttt{if } (\hat{x} + \hat{y} < 5) \texttt{ then}$
$\quad\quad \hat{x} := \hat{x} + 1;$
$\quad\quad \texttt{write}(x = \hat{x}) \}$

***(a)* Transaction T_1**

$T_2 ::= \{\hat{x} := \texttt{read}(x);$
$\quad \hat{y} := \texttt{read}(y);$
$\quad \texttt{if } (\hat{x} + \hat{y} < 5) \texttt{ then}$
$\quad\quad \hat{y} := \hat{y} + 1;$
$\quad\quad \texttt{write}(y = \hat{y}) \}$

***(b)* Transaction T_2**

$Q_{T_1} =$

$\mathcal{D}$	WT
$\{(0,0),(0,1),(0,2),(0,3),(0,4)\}$	$x := 1$
$\{(1,0),(1,1),(1,2),(1,3)\}$	$x := 2$
$\{(2,0),(2,1),(2,2)\}$	$x := 3$
$\{(3,0),(3,1)\}$	$x := 4$
$\{(4,0)\}$	$x := 5$

***(c)* The quotient set Q_{T_1} for T_1. We assume x and y are always positive. For $x + y \geq 5$, the transaction does not make any writes. The quotient set for T_2 is symmetric. (v_1, v_2) represents a database with $D(x) = v_1$ and $D(y) = v_1$.**

Figure 3.1: Example transactions and quotient sets.

3.1.3 Symbolic tables

Quotient sets for write-trace equivalence describe the relationship between a transaction's input and output. However, they are intractable to compute via the brute force method and they can be very large in the general case.

Consider the two new example transactions from Figure 3.2. These are similar to the earlier example from Figure 3.1; however, the transactions now make writes even when x and y are not between 0 and 5. Materializing the quotient sets for T_3 and T_4 is infeasible—not only is the set of database states infinite, but the number of possible write traces is also infinite.

We introduce *symbolic tables* as a concise way to represent transaction quotient sets. A symbolic table is a mapping from formulas φ, which represent a set

of database states, to write traces. The values in the write trace are either constants or functions of the values read from the database. Therefore, the write trace of a transaction can be represented as an ordered sequence of assignments where the LHS of the assignment relation is the database variable receiving the write, and the RHS is a function over database variables. We call an ordered sequence of such assignment relations, each of which corresponds to a write by the transaction, a *symbolic write trace*. Given a particular database instance D which satisfies a formula φ, the write trace produced by executing our transaction on D can be obtained by evaluating the symbolic write trace, using D to provide a valuation for the database variables.

Formally, a symbolic table for a transaction is a binary relation Q_T containing pairs $\langle \varphi_{\mathcal{D}}, \phi \rangle$, where $\varphi_{\mathcal{D}}$ is a formula in a suitably expressive first-order logic and ϕ is the symbolic write trace produced by the transaction T for each database which satisfies $\varphi_{\mathcal{D}}$. The semantics of such a pair is as follows: For every instance of the database D which satisfies the formula $\varphi_{\mathcal{D}}$, we can obtain the write trace of the transaction by applying the symbolic write trace ϕ on D. The symbolic tables for transactions T_3 and T_4 from Figure 3.2a and 3.2b are shown in Figure 3.3a and Figure 3.3b. For example, the transaction T_3 produces the symbolic write trace $[x := x + 1]$ for all databases which satisfy $x + y < 10$, so the symbolic table contains the tuple $\langle x + y < 10, [x := x + 1] \rangle$.

Typically, each execution path through the transaction code corresponds to a unique symbolic write trace, although this is not always true, for example if a transaction performs identical writes on both branches of a conditional statement.

We can extend the notion of symbolic tables to sets of transactions

T_3 ::= {$\hat{x}$:= read(x);
$\hat{y}$:= read(y);
if ($\hat{x} + \hat{y} < 10$) then
$\hat{x} := \hat{x} + 1$;
else $\hat{x} := \hat{x} - 1$;
write($x = \hat{x}$); }

***(a)* Transaction T_3**

T_4 ::= {$\hat{x}$:= read(x);
$\hat{y}$:= read(y);
if ($\hat{x} + \hat{y} < 20$) then
$\hat{y} := \hat{y} + 1$;
else $\hat{y} := \hat{y} - 1$;
write($y = \hat{y}$); }

***(b)* Transaction T_4**

Figure 3.2: Example transactions for symbolic tables.

Q_{T_1}

$\varphi_{\mathcal{D}}$	ϕ
$x + y < 10$	$[x := x + 1]$
$x + y \geq 10$	$[x := x - 1]$

***(a)* Symbolic table for T_3.**

Q_{T_2}

$\varphi_{\mathcal{D}}$	ϕ
$x + y < 20$	$[y := y + 1]$
$x + y \geq 20$	$[y := y - 1]$

***(b)* Symbolic table for T_4.**

$Q_{\{T_3,T_4\}}$

$\varphi_{\mathcal{D}}$	ϕ_1	ϕ_2
$x + y < 10$	$[x := x + 1]$	$[y := y + 1]$
$10 \leq x + y < 20$	$[x := x - 1]$	$[y := y + 1]$
$x + y \geq 20$	$[x := x - 1]$	$[y := y - 1]$

***(c)* Symbolic table for the transaction set $\{T_3, T_4\}$**

Figure 3.3: Symbolic tables for T_3 and T_4.

$\{T_1, T_2, \cdots T_k\}$. A symbolic table for a set of K transactions is a K + 1-ary relation. Each tuple in this relation is now of the form $\langle \varphi_{\mathcal{D}}, \phi_1, ..., \phi_N \rangle$, where ϕ_i is the symbolic write trace produced by transaction T_i over all databases satisfying $\varphi_{\mathcal{D}}$. Such a relation can be constructed from the symbolic tables of individual transactions as follows. Consider the cross-product of the quotient sets of all transactions. For every tuple $\langle \varphi_1, \phi_1, ..., \varphi_N, \phi_N \rangle$ in $Q_{T_1} \times Q_{T_2} ... \times Q_{T_K}$, we add a tuple $\langle \varphi_1 \wedge \varphi_2 ... \wedge \varphi_K, \phi_1, ..., \phi_N \rangle$ to the relation $Q_{\mathcal{T}}$. A symbolic table for $\{T_3, T_4\}$ is shown in Figure 3.3c.

3.1.4 Computing symbolic tables

We now explain how to use program analysis to compute symbolic tables automatically from transaction code. Our analysis is specific to a custom language we call $\mathcal{L}$; this language is simple but expressive enough to encode realistic transactions such as those in the TPC-C benchmark [3].

Let $\widehat{Var}$ be a countably infinite set of variables used temporary in the transactions (and not stored in the database). Metavariables $\hat{x}, \hat{y}, \hat{i}, \hat{j}$... range over temporary variables. There are five types of statements in $\mathcal{L}$:

- arithmetic expressions *AExp* (elements are denoted $e, e_0, e_1, ...$)
- boolean expressions *BExp* (elements are denoted $b, b_0, b_1, ...$)
- input/output commands *IO* (elements are denoted $io, io_0, io_1, ...$)
- commands *Com* (elements are denoted $c, c_0, c_1, ...$)
- transactions *Trans* (elements are denoted $T, T_0, T_1, ..$). Each transaction takes a list *Params* of zero or more integer parameters $p, p_0, p_1, ...$

The syntax for the language is shown in Figure 3.4.

Given a transaction T in the language $\mathcal{L}$, a symbolic table for T can be constructed inductively by applying the rules shown in Figure 3.5. The infix operator :: denotes the concatenation of symbolic write traces. $\varphi\{\frac{e}{x}\}$ denotes the formula obtained from φ by substituting expression a for all occurrences of x. $\phi\{e/x\}$ denotes a symbolic write trace obtained from ϕ by replacing certain occurrences of x by e. Specifically, we replace all occurrences of x that are either in the RHS of an assignment relation or on the LHS of an array assignment relation when used as an array index.

$$
\begin{array}{rl}
(AExp)\ e & ::= n \mid p \mid \hat{x} \mid e_0 \oplus e_1 \mid -e \\
(BExp)\ b & ::= \texttt{true} \mid \texttt{false} \mid e_0 \odot e_1 \mid b_0 \wedge b_1 \mid \neg b \\
(IO)\ io & ::= \hat{x} := \texttt{read}(x) \mid \texttt{write}(x = \hat{x}) \mid \\
 & \quad\ \hat{x} := \texttt{read}(a(\hat{i})) \mid \texttt{write}(a(\hat{i}) = \hat{x}) \\
(Com)\ c & ::= \texttt{skip} \mid \hat{x} := e \mid c_0; c_1 \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid io \\
(Trans)\ T & ::= \{c\}\,(P) \\
(Params)\ P & := \texttt{nil} \mid p, P \\
 & \oplus ::= + \mid * \qquad \odot ::= \ < \mid = \mid \leq
\end{array}
$$

Figure 3.4: Syntax for the language $\mathcal{L}$

Algorithmically, we can compute the symbolic table by working backwards from the final statement in each possible execution path. Each such path generates one tuple in the symbolic table.

We show how to compute the symbolic table for a variant of transaction T_3 augmented with an extra initial blind write. The left side of Figure 3.6 shows the code and the right side the computation.

First, Rule (1) initializes the symbolic table with a *true* formula representing all database states and a null write trace. Next, the symbolic table is constructed by working backwards through the code as suggested by Rule (2). We start with the last write along each execution path. (Reads or other operations performed after the *last* write cannot persist beyond the transaction.) In our example, the next rule to be applied is Rule (7). In this case, we simply add the corresponding assignment relation to the symbolic write trace. The value in the RHS of this assignment is modified in subsequent steps as we work back through the code. Because the previous statement in the program is an `if`, we then apply Rule (3) to create a copy of our running symbolic table; we use one copy for processing the "true" branch and the other for processing the "false" branch. On each of the

$$[\![T, \{\}]\!] \to [\![c, \{\langle \texttt{true}, \{\}\rangle\}]\!] \quad \text{, where } T = \{c\} \tag{3.1}$$

$$[\![c_1; c_2, Q]\!] \to [\![c_1, [\![c_2, Q]\!]]\!] \tag{3.2}$$

$$\left[\!\!\left[\begin{array}{l} \texttt{if } b \texttt{ then } c_1 \\ \texttt{ else } c_2 \end{array}, Q \right]\!\!\right] \to \left\{ \begin{array}{l} \{\langle b \wedge \varphi, \phi\rangle \mid \langle \varphi, \phi\rangle \in [\![c_1, Q]\!]\} \cup \\ \{\langle \neg b \wedge \varphi, \phi\rangle \mid \langle \varphi, \phi\rangle \in [\![c_2, Q]\!]\} \end{array} \right\} \tag{3.3}$$

$$[\![(\hat{x} := e), Q]\!] \to \left\{\langle \varphi\{\frac{e}{\hat{x}}\}, \phi\{e/\hat{x}\}\rangle \mid \langle \varphi, \phi\rangle \in Q\right\} \tag{3.4}$$

$$[\![\texttt{skip}, Q]\!] \to Q \tag{3.5}$$

$$[\![\hat{x} = \texttt{read}(x), Q]\!] \to \left\{\langle \varphi\{\frac{x}{\hat{x}}\}, \phi\{x/\hat{x}\}\rangle \mid \langle \varphi, \phi\rangle \in Q\right\} \tag{3.6}$$

$$[\![\texttt{write}(x = \hat{x}), Q]\!] \to \left\{\langle \varphi\{\frac{\hat{x}}{x}\}, [x := \hat{x}] :: \phi\{\hat{x}/x\}\rangle \mid \langle \varphi, \phi\rangle \in Q\right\} \tag{3.7}$$

$$[\![\hat{x} := \texttt{read}(a(\hat{i})), Q]\!] \to \left\{\langle \varphi\{\frac{a(\hat{i})}{\hat{x}}\}, \phi\{a(\hat{i})/\hat{x}\}\rangle \mid \langle \varphi, \phi\rangle \in Q\right\} \tag{3.8}$$

$$[\![\texttt{write}(a(\hat{i}) = \hat{x}), Q]\!] \to \left\{\langle \varphi\{\frac{a\{\hat{i} \leftarrow \hat{x}\}}{a}\}, [a(\hat{i}) := \hat{x}] :: \phi\{\hat{x}/a(\hat{i})\}\rangle \mid \langle \varphi, \phi\rangle \in Q\right\} \tag{3.9}$$

Figure 3.5: Rules for constructing a symbolic table for transaction T; Q is the running symbolic table.

two paths, we encounter an assignment statement to variable $\hat{x}$. Since $\hat{x}$ appears on the RHS in our running symbolic write traces, we apply Rule (4) to update ϕ appropriately. Continuing through the code, we next encounter a read. This relates the temporary variable $\hat{y}$ to the object y in the database, and we capture that information by applying the substitution required by Rule (6). We apply the same rule again to process $\texttt{read}(x, \hat{x})$. Special handling is required for variables which are read after being written; the value written needs to propagate forwards through the code so that subsequent reads are aware of it. For this reason, Rule (7) requires substitutions in the running symbolic write trace. This is seen in action as we process the first statement of our transaction—the blind write of 6 to database object x. The value 6 is substituted for x in our formulas φ and write traces ϕ as required for correctness. This completes the computation.

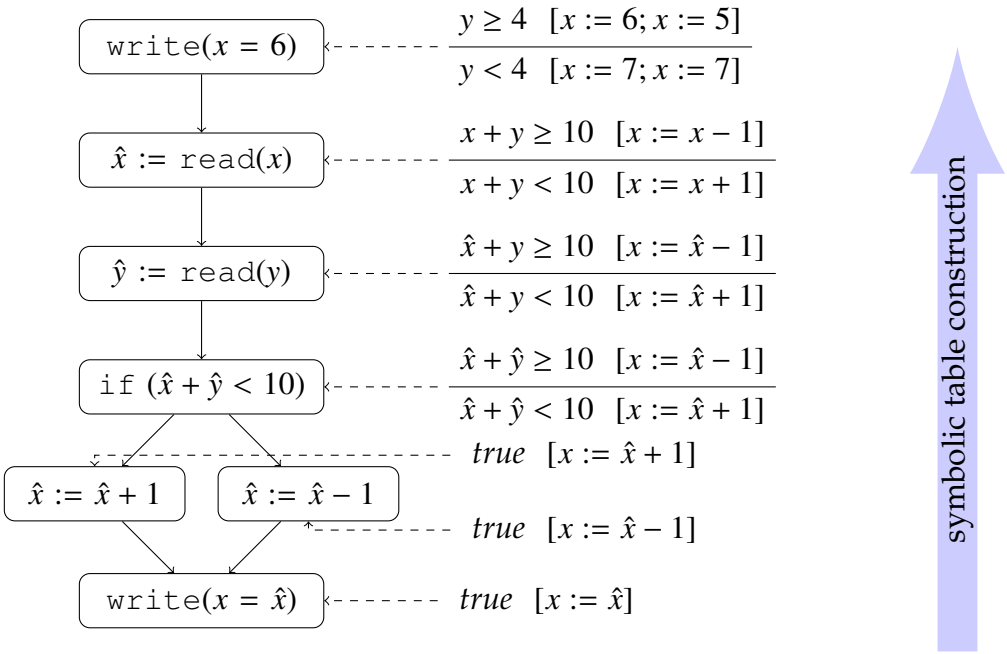

Figure 3.6: Example of symbolic table construction

Note that each tuple in the symbolic table corresponds to a unique execution path in the transaction, and thus a given database instance D may only satisfy a single formula φ in Q_T. Also note that once we are done, the symbolic table only references database objects, constants and input parameters rather than temporary variables.

3.1.5 Language expressiveness and extensions

Although $\mathcal{L}$ is not Turing-complete and in particular has no loops, it is expressive enough to encode many realistic database transactions. If we assume a bound on the maximum number of tuples in any relation, we can express bounded (foreach) iteration and we are able to encode `SELECT-FROM-WHERE` clauses and in fact all five transactions in TPC-C [3]. This encoding may lead to

large $\mathcal{L}$ programs, but we can apply compression and optimization techniques to mitigate this problem, as discussed in Section 3.4.

Further language extensions are possible, e.g. to support unbounded iteration or complex data structures. These extensions are compatible with our analysis, as long as we add appropriate rules for symbolic table computation and understand that the symbolic table formulas φ may need a more expressive logical language.

3.2 The Homeostasis Protocol

This section gives the formal foundation for our homeostasis protocol that uses semantic information to guarantee correct disconnected execution in a distributed system. We begin by introducing a simple system model (Section 3.2.1) and several preliminary concepts (3.2.4) and then present the full protocol (3.2.5). The material in this section applies to transactions in any language, not just $\mathcal{L}$ as introduced in Section 3.1.4.

3.2.1 Distributed system model

Assume the system has K sites, each database object is stored on a single site, and transactions run across several sites. Formally, a distributed database is a pair $\langle D, Loc \rangle$, where D is as before and $Loc : Var \to \{1, \ldots, K\}$ is a mapping from variables to the site identifier where the variable value is located. Each transaction T_j runs on a particular site i. Formally, there is a *transaction location function* ℓ from transactions to sites such that for every T_i, $\ell(T_i)$ is a site identi-

fier. A schedule represents a concrete execution of a set of transactions in our distributed system model.

Definition 3.2.1 (Schedule) *Assume we have a set of transactions $T_1, T_2, \cdots T_k$. A schedule $(S, \leq)$ is a partially ordered set of operations performed by these transactions under some partial order $\leq$ and transaction location function ℓ. S, ℓ and $\leq$ must satisfy the following properties:*

- *each operation is tagged with the identifier of its transaction*
- *for every pair of operations o, o' by transaction T, if o occurs before o' in T then $o \leq o'$ in S*
- *for every pair of operations on the same object $o_i(x)$ and $o_j(x)$ such that at least one operation is a write, we have either $o_i(x) \leq o_j(x)$ or $o_j(x) \leq o_i(x)$.*
- *for every pair of operations at the same site - i.e. for o_i and o_j where $\ell(T_i) = \ell(T_j)$, we have either $o_i \leq o_j$ or $o_j \leq o_i$.*

The operations o, o' mentioned in the definition are either reads or writes, as is standard in concurrency theory. We denote schedules as just S when the order $\leq$ is clear from context.

We will introduce the homeostasis protocol in the setting of a distributed system where the following crucial assumption holds.

Assumption 3.2.2 (All Writes Are Local) *If T_j runs on site i, it may only perform write operations on variables x such that $Loc(x) = i$, that is, all writes are local to the site on which the transaction runs. More formally, if we have a schedule with an operation $w_j(x)$, then we must have $\ell(T_j) = Loc(x)$.*

3.2.2 Homeostasis in the replicated setting

Our distributed model under this assumption is powerful enough to encode systems that use replication. We now show how to adapt our system model and protocol to a setting where objects are replicated on multiple sites. Formally, the function *Loc* now maps variables to *nonempty subsets* of $\{1, 2, \cdots K\}$.

One way to encode replicated execution in our distributed model is as follows. Given an object x, each site maintains a "snapshot" x_s of x (and this snapshot is the same across all sites). When a transaction at site i wishes to write to x, it does not overwrite the snapshot, rather, it writes a suitable "delta" value v to a new local database object dx_i. v is chosen so that $v + x_s$ is the desired new value of x. This can be repeated if the site wishes to update x again; the old dx_i is not overwritten but incremented as appropriate.

While site i is proceeding as above, other sites $j \in Loc(x)$ will be doing the same and maintaining their own dx_j objects. At any time in the system, the current correct value of x is thus $x_s + \sum_{k \in Loc(x)} dx_k$. This means that while *writes* to x involve only a local update of the appropriate dx object, *reads* of x require reading everyone's dx objects for correctness. This setup fits our distributed system model and obeys Assumption 3.2.2.

Given a transaction program, it is straightforward to transform it to use dx objects as required for the above encoding, and this transformation can be performed automatically by a compiler. In this transformation, it is possible to recognize that certain updates do not actually require reading remote sites' deltas for correctness; for example, an integer increment operation has the same delta (namely, 1), regardless of the current value of the integer. Thus an increment op-

eration on x can directly be compiled into an increment to the local dx, avoiding remote reads.

When running the homeostasis protocol in the replicated case, we have treaties on the delta objects dx. As long as the treaties are respected, site i can avoid reading dx_j for $j \neq i$. At the end of each round of the protocol, it is useful—though not necessary for correctness—to update the local snapshots x_s for each object x to the real current value of x and reset the delta objects to 0.

If we extend the transaction language beyond arrays of integers to support other data types, it is still possible to perform the above "delta object" transformation in certain cases. To achieve this, we need the data type in question to come with a suitable *merge function* that is the equivalent of addition in the integer case. Under this merge function, elements of the data type must form an associative Abelian group. For example, we can apply the delta object transformation to multisets (bags) of integers, using bag union as the merge function.

If the data type we wish to use does not come with a suitable merge function, the delta object transformation cannot be used and it is necessary to synchronize on every update to a replicated object of this type. However, existing research has shown that there are numerous commutative replicated data types which do come with the merge functions we need [96]. Thus in practice we expect the delta object transformation to apply in the vast majority of cases.

3.2.3 Preliminaries

This section contains various technical preliminaries needed to develop the proofs presented in the rest of this chapter.

We assume that transactions read and write the same database objects regardless of what database instance they run on. If this is not true, we insert *dummy reads and writes* into transaction code to ensure the (ordered) sequence of reads and writes performed by each transaction is the same regardless of the starting database.

If a particular schedule S is executed on a starting database D, every transaction T in S will have a concrete read trace and write trace that represents the literal values read and written by T. We can thus define the read and write traces of T in S for a given D. The read trace of T in S on D, denoted $RT_D(T, S)$ is the concrete ordered sequence of values read by T in S, and analogously for the write trace, which is denoted $WT_D(T, S)$.

Definition 3.2.3 (Totally Ordered Projection) *Given a schedule S, a totally ordered projection of S is any total ordering of the operations in S that preserves the partial ordering imposed by S.*

Definition 3.2.4 (Serial Schedule) *A schedule S is serial if $\leq$ is a total order and it contains no interleavings, i.e. for all pairs of transactions T_i, T_j, S orders all operations of T_i before all operations of T_j or vice versa.*

Definition 3.2.5 (View equivalence) *Let S and S' be two totally ordered schedules. These are view-equivalent if all three of the following properties hold:*

- *If transaction T reads the initial value of some object x in S, it also reads the initial value of x in S' and vice versa*
- *If transaction T performs the final write to some object x in S, it also performs the final write on x in S' and vice versa*
- *If transaction T reads an object x in S and the immediate previous write to x was by T', we say T* reads from *T' in S. If T reads from T' in S, it also reads from T' in S' and vice versa.*

Lemma 3.2.6 *Given a schedule S, all totally ordered projections of S are view-equivalent to each other.*

Proof 3.2.7 *This is immediate from the definition of a schedule as we require all reads on a given object totally ordered with respect to any writes.*

Definition 3.2.8 (View serializability) *A schedule S is view-serializable if there is some serial schedule $\hat{S}$ involving the same transactions with the property that all totally ordered projections of S are view-equivalent to $\hat{S}$.*

By the lemma above, all the totally ordered projections are view-equivalent to each other, so when proving view serializability it suffices to prove view-equivalence of one totally ordered projection to the serial schedule.

3.2.4 LR-slices, translations and treaties

As explained in the introduction, the idea behind the homeostasis protocol is to allow transactions to operate *disconnected*, i.e. without inter-site communication,

T_5 ::= {$\hat{x}$:= `read`(x);
`if` ($\hat{x} > 0$) `then` $\hat{y} := 1$;
`else` $\hat{y} := -1$;
`write`($y = \hat{y}$); }

***(a)* Transaction T_5**

T_6 ::= {$\hat{x}$:= `read`(x);
$\hat{y}$:= `read`(y);
`if` ($\hat{y} = 1$) `then` $\hat{z} := (\hat{x} > 10)$;
`else` $\hat{z} := (\hat{x} > 100)$;
`write`($z = \hat{z}$); }

***(b)* Transaction T_6**

Figure 3.7: Example transactions for protocol. x is remote for both transactions, while y and z are local.

and over a database state that may be inconsistent, as long as appropriate *treaties* are in place to bound the inconsistency and guarantee correct execution. Our protocol relies on the crucial assumption that all transaction code is known in advance. This assumption is standard for OLTP workloads.

In our model, the desired "disconnected" execution of transactions means that we do not perform remote reads—i.e., a transaction only performs reads on the site where it resides. That is, instead of running a transaction that runs on site i but reads values from multiple other sites, we run a modified transaction that only "really" reads the values that are local to site i. For values resident at other sites, the transaction reads a locally available older *snapshot* of these values.

Consider transaction T_5 from Figure 3.7. Instances of this transaction write to y, so they must run on the site where y resides. For the sake of the example, assume that x resides on another site. We could avoid the remote read of x by taking a current snapshot of x, say 10, and inlining it into the code. Thus we create a *translation* of the transaction where the operation $\hat{x}$:= `read`(x) is replaced by $\hat{x} := 10$; of course, this code can be optimized further to propagate constants. Instances of our translated transaction produce the correct writes on y

even if the value of x should change on the remote site, as long as that site never allows x to go negative. Thus, if we can set up a *treaty* whereby the system commits to keeping x positive, we can safely run instances of the translated transaction.

To formalize the intuition provided by the above example, we explore how the local and remote reads of a transaction affect its write trace. Consider transaction T_6 from Figure 3.7. Assume that y and z are local and x is remote. The write trace of T_6 depends on both local and global state. Suppose we begin with a database containing $x = 20, y = 1$. We can compute a translated transaction by replacing $\texttt{read}(x, \hat{x})$ with $\texttt{read}(20, \hat{x})$. Since $y = 1$, It seems the appropriate treaty for the system would be to maintain $x > 10$. However, this is not sufficient to guarantee correct behavior. Suppose that x changes to 200 (which is allowed per our treaty), and in addition some other transaction changes y to 3. Then the database has $x = 200, y = 3$. If we run an instance of the original transaction, z is set to *true*. However, if we run an instance of the translated transaction, z is set to *false*. The translation is no longer correct, although the remote site respected its treaty. The problem was due to the local change that set y to 3. This local change had an impact on *the manner in which the remote state affects the transaction's writes*.

The next few definitions formalize this behavior precisely.

Definition 3.2.9 (Local-remote partition) *Given a database D, a* local-remote partition *is a boolean function p on database objects in D. If $p(x) = true$ we say x is* local *under p, otherwise it is* remote.

Given a local-remote partition p, we can express any database D as a pair (l, r) where l is a vector of values of all x that are local and r is a vector of values

of all y that are remote under p.

For example, if we have the database $D = \{x = 1, x' = 2, y = 3, y' = 4\}$, and we define $p(x) = p(x') = true$, $p(y) = p(y') = false$, then we can express D as $(\langle 1, 2\rangle, \langle 3, 4\rangle)$.

Definition 3.2.10 (Local-remote slice) *Let T be a transaction and p a local-remote partition. Consider a pair L, R, where each $l \in L$ is a vector of values to be assigned to local database objects, and each $r \in R$ is a vector of values to be assigned to the remote objects. Then L, R define a set of databases $\{(l, r) \mid (l \in L, r \in R)\}$. (L, R) is a* local-remote slice *(or LR-slice for short) for T if the following holds: $\forall l \in L, \forall r, r' \in R \;\; WT(T, (l, r)) = WT(T, (l, r'))$.*

Intuitively, for any database in the LR-slice, we know that as long as the remote values stay in R, then the write trace of the transaction is determined *only* by the local values.

Example 3.2.11 *Consider our transaction T_6 from above, with y being local and x remote. For notational clarity, we omit z and list the permitted values for y first, then those for x. One LR-slice for T_6 is $(\{1\}, \{11, 12, 13\})$, another is $(\{1\}, \{11, 12, 13, 14\})$, and yet another is $(\{2, 3, 4\}, \{0, 1, 2, 3\})$.*

There is no maximality requirement in our definition of *LR*-slices, as the above example shows. In addition, there is a complex relationship between *LR*-slices and write-trace equivalent sets of databases. Two databases that are not write-trace equivalent for a transaction T may be in the same *LR*-slice for T, and conversely it is possible to have two databases that are write-trace equiva-

lent for T but where no *LR*-slice for T can contain them both. Next, we formalize translations.

Definition 3.2.12 (Translation) *Given a transaction T, a database D and a local-remote partition p, the* translation *T' of T using D and p is the program generated from T by the following process. Consider every read operation $\hat{x} :=$ `read`(x) in T where x is remote under p. Replace it with an operation $\hat{x} := v$, where $v = D(x)$.*

The following Lemma relates LR-slices and translations to each other. It formalizes the intuition that a program translation is correct (i.e. equivalent to running the original program) as long as the database is in the same LR-slice as the state on which the translation was computed.

Lemma 3.2.13 *Assume the following are given: $p, T, D = (l, r)$ and some LR-slice (L, R) with the property that $l \in L, r \in R$. Let T' be the translation of T on D. Let D' be any other database in the same LR-slice. Then $WT(T, D') = WT(T', D')$.*

Proof 3.2.14 *Let the D' in question be (l', r'). We know several useful facts.*

1. *$WT(T, (l', r')) = WT(T, (l', r))$ from the definition of LR-slices.*
2. *$RT(T, (l', r)) = RT(T', (l', r))$: because of the way we generated translations, a transaction and its translation have the same read trace if the remote values that were inlined into the translation are still unchanged in the underlying database.*
3. *$WT(T, (l', r)) = WT(T', (l', r))$: this follows from the previous point and from the fact that T and T' run exactly the same code except for inlining some reads.*
4. *$RT(T', (l', r)) = RT(T', (l', r'))$ because T' does not perform any remote reads*

5. $WT(T', (l', r)) = WT(T', (l', r'))$ *which follows directly from the previous point.*

Combining 1, 3 *and* 5 *above gives us the desired* $WT(T, (l', r')) = WT(T', (l', r'))$ *and we are done.*

Next, we define global treaties.

Definition 3.2.15 (Global Treaty) *A* global treaty Γ *is a subset of the possible database states* $\mathcal{D}$ *for the entire system. The system* maintains *the treaty by disallowing any write operations that would produce a state not contained in* Γ.

For our purposes, it will be useful to have treaties that require the system to remain in the same *LR*-slice as the original database. Under such a treaty, by Lemma 3.2.13, it will be correct to run the translated transactions instead of the ones that perform remote reads. We formalize the treaty property we need as follows.

Definition 3.2.16 (Valid Global Treaty) *Let* $\{T_i\}_{i \in I}$ *a set of transactions for some index set* I. Γ *is a* valid global treaty *if the following is true. For every* T_i *with associated local-remote partition* p_i, *let* $L = \{l \mid (l, r) \in \Gamma\}$ *and* $R = \{r \mid (l, r) \in \Gamma\}$. *We must have that* (L, R) *is a LR-slice for* T_i.

Finally, we define one more technical but crucial and desirable property of transaction executions under a global treaty.

Definition 3.2.17 (Local Consistency) *Consider* T *that runs in a system, potentially interleaved with other transactions, and a valid global treaty* Γ. *The execution is* locally

consistent *with* Γ *if at all intermediate points in the execution the following holds. Denote by* σ_l *the current local read trace of* T*, i.e. the trace that contains only reads of local objects. There must exist some* $D' \in \Gamma$ *such that running* T *on* D' *to the same point in its execution would produce precisely* σ_l *as the local read trace.*

To understand the above property and why we need to enforce it, consider this example:

Example 3.2.18 *Let* T_7 *be a transaction that reads the values of local objects* x *and* y *into* $\hat{x}$ *and* $\hat{y}$ *respectively, and then performs different writes based on whether* $(\hat{x} = 1) \vee (\hat{y} = 1)$ *holds. Suppose the initial* D *has* $x = 0, y = 1$*, and the treaty* Γ *allows only database states where* $(x = 1) \vee (y = 1)$ *is* true*. However, suppose we have an execution as follows, where* T_7*'s reads of* x *and* y *interleave with writes by two other transactions* 8 *and* 9*:*

$$r_7(x, \hat{x})\ w_8(1, x)\ \ w_9(0, y)\ \ r_7(y, \hat{y})$$

This execution preserves the property that $(x = 1) \vee (y = 1)$ *is always* true *and is view serializable, yet* T_7*'s reads see* $x = 0, y = 0$*, so* T_7 *has "seen" a local database state not permitted by* Γ*.*

This property can be enforced in a variety of ways, for example using snapshot isolation.

3.2.5 Homeostasis protocol

We next present our protocol to guarantee correct execution of transactions without remote reads. Assume our set of transactions is $\{T_i\}_{i \in I}$. Each (instance

of) T_i runs on some site j and this induces a local-remote partition p_i; from the perspective of (instances of) T_i running at site j, all values on j are local and all others are remote.

The protocol proceeds in rounds. Each round has three phases: treaty generation, normal execution and cleanup.

Treaty generation: In this phase, the system uses the current database D to generate a valid global treaty Γ, where $D \in \Gamma$, and computes a translation T_i' for each T_i using p_i.

Normal execution: In this phase, the system runs instances of the translated transactions T_i'. Transactions running on a single site may interleave; locally at each site, the system uses any concurrency control algorithm that provides the two following guarantees:

1. classic view-serializability, and
2. *local consistency* with Γ for each T (Definition 3.2.17).

If a running transaction ever tries to make a write that would lead to a database $D' \notin \Gamma$, then the cleanup phase begins.

Cleanup: Suppose T' is the transaction that triggered the cleanup phase. In a multi-site system, several transactions may try to make a treaty-violating write at the same time; we rely on a suitable voting algorithm to choose one "winner" among these. We abort any "loser" transactions that also tried to violate the treaty and wait for all transactions other than T' to terminate (if, due to the local concurrency control algorithm used, some transactions are blocking/waiting for T', they are aborted as well). We then allow the write by T' to proceed and

T' to terminate, and we start a new round of the protocol, taking care to rerun any transactions that were aborted in the cleanup phase.

We now argue that the protocol above is correct. It is not trivial to capture a good notion of "correctness"; we are after all reasoning about transactions that execute on *inconsistent data*, which is not a common setting in classical concurrency theory.

Intuitively, a round of the above protocol produces a schedule (or execution) of some translated transactions. We show that there is some "corresponding" execution of the original untranslated transactions which produces the same writes and is view serializable. We use this as our notion of correctness; the idea is that the latter (view-serializable) execution is clearly correct, and under our requirement that both produce the same writes an external observer could not distinguish it from the former.

Theorem 3.2.19 *Fix a starting database D and distributed schedule $(S', \leq_{S'})$ produced by a single round of the protocol above. There is a schedule $(S, \leq_S)$ with the following properties:*

- *For every operation o'_i in S' performed by an instance of a translation T'_i, S contains a* corresponding *operation o_i performed by T_i. S contains no additional operations.*
- *If $o_i \leq_{S'} o_j$ then $o_i \leq_S o_j$*
- *For every T in S and corresponding T' in S', the write traces of T and T' are the same.*
- *S is view serializable.*

Proof 3.2.20 *We construct $(S, \leq_S)$ as follows. We begin by replacing each operation in S' by the corresponding operation in the untranslated transaction. Most operations stay the same, but any translated reads in S' of the form $\hat{x} := v$ are replaced with the original reads $\hat{x} :=$ `read`(x).*

We next construct the new ordering $\leq_S$. We start with $\leq_{S'}$, and augment this by totally ordering remote reads on each object x with respect to all writes on x. We proceed as follows.

First, we construct a total ordering $\prec$ on the transactions *in S' (and by extension on the transactions in S).*

For each site, we know local view serializability is guaranteed by the protocol. Therefore, there is at least one total ordering of the transactions at that site which is guaranteed to be view equivalent to the interleaved execution that actually occurred. Pick any such ordering for each site. Now choose a global total ordering on the transactions that is consistent with each of the local total orders just chosen, and where the transaction whose write caused a treaty violation (and triggered the next protocol round) is the very last transaction. This completes the construction of $\prec$.

We are now ready to order remote reads appropriately in S. Let T_i be any transaction that performs a remote reads on site k. We order these reads in a way such that:

- *these remote reads may not interleave with any other operations on site k by any other transaction, and*
- *the ordering respects $\prec$ as defined above.*

More formally, suppose T_i performs a remote read $\hat{x} :=$ `read`(x) where $Loc(x) = k$. Denote this read as $r_i(x)$ for clarity. Then for every operation – i.e., read or write – $o_j(z)$

by some other transaction T_j having $Loc(z) = k$, we set $o_j(z) \leq_S r_i(x)$ if $T_j \prec T_i$ and $r_i(x) \leq_S o_j(z)$ otherwise.

This completes the construction of $(S, \leq_S)$. By construction, $(S, \leq_S)$ satisfies the first two properties the Theorem requires.

We now show the third property, namely for every T in S and corresponding T' in S', $WT_D(T, S) = WT_D(T', S')$

This is an inductive proof using the partial order on all write operations in S' and S (by construction of S, this partial order on writes is the same in S and S').

We start with any write operation in S' that does not have any predecessor write operations. This operation `write`*$(x = v')$ is performed by some transaction T'; the corresponding* `write`*$(x = v)$ is performed by T in S. The read traces of T and T' so far are* identical. *It follows that $v = v'$ as desired.*

Now consider any write `write`*$(x = v')$ by T' in S',* `write`*$(x = v)$by T in S, and assume for our inductive hypothesis that all write operations that are predecessors of this write had the same value in both S and S'. Note that we can no longer assume that the read trace of T and T' so far are identical, because T' is a translation and "reads" an old snapshot of the remote values which may have changed since the beginning of the execution. We do know that the* local *portion of the read traces is identical, however (we are using the inductive hypothesis here as we know that all operations on a single site are totally ordered). By the protocol's local consistency guarantee, there is some database $(l, r) \in \Gamma$ such that the local portion of the read trace (so far) is consistent with this database.*

The remote portion of the traces may be different; the real reads by T may see dif-

ferent values than the fake "reads" by T'. Observe that if T should ever read the same value twice it will get the same answer (by our requirement on $\leq_S$ that a transaction's remote reads are not allowed to interleave with anyone else's writes), and similarly for T' (because the two "fake read" operations will have the same snapshot values inlined). Thus we can blur the distinction between the remote portion of the read traces *so far and the remote database* state *that was seen by each transaction so far. The remote database state seen by the real reads of T is the actual remote portion of the database r_a on which its remote reads are evaluated (all of T's reads are evaluated on the same r_a, again by our construction of $\leq_S$). The remote database state "seen" by the fake reads of T' is r_b, which is the remote values of the original database D (as those were inlined into the translation). By the protocol, it is guaranteed that r_a and r_b belong to the same LR-slice for T. (The only writes that may take the system into a different LR-slice are the writes of the last transaction in the protocol round, and those writes are not read by any other transaction in the round because of the way the cleanup phase proceeds.)*

Putting it all together, the read trace so far of T is consistent with the database (l, r_a), and the read trace so far of T' is consistent with the database (l, r_a). Both of these databases belong to the same LR-slice for T, so it follows that the next upcoming write by T and by T' must produce the same value as well.

It remains to show the last property, that our new schedule S is view serializable. Let $\hat{S}$ be a serial schedule constructed from S by using the total transaction ordering $\prec$ that we previously defined. Let S'' be any totally ordered projection of S. We need to show that S'' and $\hat{S}$ are view equivalent.

The first and second requirements for view equivalence follow from the fact that we guarantee view serializability locally and that only local writes are permitted. The third follows by our construction of $\leq_S$ and from the local view serializability guarantee. If

T_j reads from T_i, T'_j read is either local - in which case the reads-from relationship is preserved by the local view serializability guarantee - or remote, in which case it is preserved by our construction of $\leq_S$ to be consistent with $\prec$.

The first condition captures the fact that S is the "corresponding" schedule to S'. The second states that any operations that are ordered in a particular way by S' remain ordered in the same way by S. S will generally contain more ordering constraints than S', because S contains remote reads that were absent in S'; those reads must be ordered with respect to writes on the same objects. The remaining conditions formalize the correctness of our protocol.

3.3 Generating treaties

Having laid the theoretical groundwork for the homeostasis protocol, we move to practical considerations. The crucial questions are how the global treaty is generated and how it is enforced; we address the former in Section 3.3.1 and the latter in 3.3.2 and 3.3.3.

As a running example, we use transactions T_3 and T_4 from Figure 3.2. Assume that x and y are on different sites, T_3 runs on the site where x resides, and T_4 runs on the site where y resides.

For ease of exposition, we temporarily assume that all symbolic write traces are functions of local database objects only; formally:

Assumption 3.3.1 *For all transactions T, for all symbolic write traces ϕ_T in the symbolic table for T, for or any variable x that occurs on the RHS of any assignment in ϕ_T,*

$Loc(x) = \ell(T)$.

This assumption holds for T_3 and T_4; we explain how to lift it in Section 3.3.4.

3.3.1 Finding a global treaty

The first task is to generate a valid global treaty (per Definition 3.2.16). A naïve but valid global treaty would restrict the database state to the initial D, i.e. require renegotiation on every write. This reduces the homeostasis protocol to a distributed locking protocol; of course, we hope to do better. Intuitively, a good global treaty is one that maximizes the expected length of a protocol round—i.e. one that has a low probability of being violated.

To find a good global treaty, we use the symbolic tables we computed for our transaction workload. At the start of a protocol round, we know the current database D, the set $\mathcal{T}$ of all transactions that may run in the system, and the symbolic table $Q_{\mathcal{T}}$ for $\mathcal{T}$, as shown in Figure 3.3c for our example set $\{T_3, T_4\}$.

Based on this information, computing a valid global treaty is straightforward: pick the unique formula φ_Γ in $Q_{\mathcal{T}}$ satisfied by D. For T_3 and T_4, and assume that the initial database state is $x = 10$ and $y = 13$. Then φ_Γ is $x + y \geq 20$ (third row of $Q_{\mathcal{T}}$ in Figure 3.3c).

Lemma 3.3.2 *Let Γ be the set of all databases satisfying φ_Γ. Under assumption 3.3.1, Γ is a valid global treaty.*

Proof 3.3.3 *We begin by introducing a useful piece of notation. For a database D and site i, define the projection of D on site i, $\Pi_i(D)$, as $\{x \in D \mid Loc(x) = i\}$.*

Consider any database $D \in \Gamma$ and $T \in \mathcal{T}$. Let $\ell(T) = i$. Let $l = \Pi_i(D)$. Consider the set of databases in Γ which have l as their projection on site i, i.e., $\mathcal{D}_l = \{D \mid \Pi_i(D) = l \wedge D \in \Gamma\}$. To prove that Γ is a valid global treaty, we need to show that T produces the same write trace for every database in $\mathcal{D}_l$. Since every database in $\mathcal{D}_l$ satisfies φ, the symbolic write trace ϕ_T produced by T is the same. Furthermore, under Assumption 3.3.1, the symbolic write trace does not refer to remote variables and therefore applying ϕ_T on any $D \in \mathcal{D}_l$ produces the same write trace. This is because the write trace is completely determined by l which by definition is the same for all $D \in \mathcal{D}_l$. Since we chose the initial database and transaction arbitrarily, Γ is in fact a valid LR-slice for all transactions. Therefore, Γ is a valid global treaty.

Γ is not in general the *largest possible* valid treaty—pathological examples are possible where it could be extended. However, φ_Γ can be computed very quickly from the symbolic table $Q_\mathcal{T}$ so it is a good global treaty for our purposes. In fact, the symbolic table can be precomputed once and reused at each treaty computation.

Our approach does not depend on the way that symbolic tables are computed, or even on the language of the transactions and the symbolic tables. If we wish to work with a language richer than $\mathcal{L}$, this is possible as long as we can compute symbolic tables.

From now on, we will blur the distinction between Γ and φ_Γ, and work with an intensional notion of a treaty as a first-order logic formula that represents a set of possible database states.

3.3.2 Local treaties

If a global treaty requires inter-site communication for enforcement, this destroys the performance advantage to be gained through the homeostasis protocol. Therefore, we need to enforce global treaties by checking suitable *local treaties* at each site.

More formally, given the global treaty φ_Γ and a current database D that satisfies φ_Γ, we need to factorize it into a set of local treaties $\langle \varphi_{\Gamma_1}, \varphi_{\Gamma_2}, ..., \varphi_{\Gamma_K} \rangle$. The local treaty φ_{Γ_i} is a first-order logic formula corresponding to the site i and uses as free variables only the database objects on site i. The conjunction of all the local treaties should imply the global treaty.

$$\forall D'. \bigwedge_{1 \le i \le K} \varphi_{\Gamma_i}(D') \rightarrow \varphi_\Gamma(D') \tag{H1}$$

We also require each local treaty to hold on the original database D; this excludes trivial factorizations such as setting all local treaties to *false*. Formally, for each i $(1 \le i \le K)$ we require the following:

$$\varphi_{\Gamma_i}(D) \tag{H2}$$

In the general case, finding local treaties is difficult. The requirements in H1 and H2 create a set of Horn clause constraints $\mathcal{H}$. For an undecidable fragment of first-order logic (such as non-linear arithmetic over integers) finding the solution to $\mathcal{H}$ is undecidable, so the local treaties may not exist. For some first-order theories such as the theory of arrays, solving $\mathcal{H}$ is decidable but the local treaties are not necessarily quantifier-free. However, for certain theories such as linear

integer arithmetic, the existence of solutions to $\mathcal{H}$ is decidable, solutions can be effectively computed and the solutions are guaranteed to be quantifier-free. These fragments of first-order logic are the theories that have the quantifier-free Craig's interpolation property. For a more elaborate discussion of solving Horn clause systems we refer the reader to [94].

3.3.3 Computing good local treaties

We present a method for computing local treaties that is not precise – i.e. it may not find the "optimal" local treaties that maximize protocol round length. However, it always works; in the worst case it degenerates to requiring synchronization on every write. It also allows us to optimize the local treaties based on a model of the expected future transaction workload. Such a model could be generated dynamically by gathering workload data as the system runs, or in other ways. The length of time spent in the optimization is a tunable knob so we can trade off treaty quality versus treaty computation time. This is important because local treaties need to be computed at the start of every protocol round; we do not want the cost of treaty computation to wipe out the savings from reduced communication during the protocol round itself.

There are three steps in our algorithm; preprocessing the global treaty, generating *templates* for the local treaties, and running an optimization to instantiate the templates into actual local treaties.

We begin by preprocessing the treaty φ_Γ to obtain a stronger but simpler treaty that is a conjunction of *linear constraints*. A linear constraint is an expres-

sion of the form

$$\left(\sum_i d_i x_i\right) \odot n$$

where the d_i and n are integers, the x_i are variables and $\odot$ is either $=$, $<$ or $\leq$. The preprocessing causes a loss of precision in that we are now enforcing a stronger treaty than the original. However, linear constraints and their variants arise frequently as global treaties in OLTP workloads and feature prominently in related work [14, 58, 106], thus we expect to handle most real-world global treaties with minimal precision loss.

The preprocessing of φ_Γ involves identifying all subexpressions ψ that prevent it from being a conjunction of linear constraints, for example ψ may be a negated boolean expression $\neg b$. For each such expression, we replace it within φ_Γ by the actual value of ψ on D. In addition, if the variables in ψ are $x_1, x_2, \cdots x_n$, we transform φ_Γ to $\varphi_\Gamma \wedge \bigwedge_i (x_i = D(x_i))$. Intuitively, any variables involved in the subexpression have their values fixed to the current ones. It is clear that the transformed φ_Γ implies the original one, and it is thus a valid global treaty as well.

The next step is to create *local treaty templates*; these contain fresh *configuration variables* that are instantiated later. We generate a template for each site k using the following method. Start with φ_Γ and proceed clause by clause. Given a clause of the form

$$\left(\sum_i d_i x_i\right) \odot n,$$

we replace it by

$$\left(\sum_{Loc(x_i)=k} d_i x_i + c_k\right) \odot n$$

where c_k is a fresh configuration variable.

With our running example transactions T_3 and T_4, initial database state $x = 10$ and $y = 13$ and global treaty $x + y \geq 20$, this process would yield the local treaty templates. $\varphi_{\Gamma_1} := (x + c_y \geq 20)$ and $\varphi_{\Gamma_2} := (c_x + y \geq 20)$, where c_x and c_y are configuration variables.

The following theorem states there is always at least one assignment of values to configuration variables that yields actual local treaties. We call any such assignment a *valid treaty configuration*.

Theorem 3.3.4 *Given a global treaty, a starting database D, and a set of local treaty templates generated using the above process, there always exists at least one way to assign values to the configuration variables that satisfies H1 and H2.*

Proof 3.3.5 *We generate the assignment of values to configuration variables as follows. Consider a configuration variable c_k introduced to replace a reference to some variable x. If c appears in an equality, i.e. a clause of the form*

$$\sum_{Loc(x_i)=k} d_i x_i + c_k = n$$

Then we replace it by $\sum_{Loc(x_j)\neq k} d_j D(x_j)$. This is an integer which can be computed by evaluating the expression above on the original database D.

Otherwise, if c_k appears in an inequality of the form

$$\sum_{Loc(x_i)=k} d_i x_i + c_k \leq n$$

We replace it with $-\sum_{Loc(x_i)=k} d_i D(x_i) + n$. *Thus the expression simplifies to*

$$\sum_{Loc(x_i)=k} d_i x_i - \sum_{Loc(x_i)=k} d_i D(x_i) + n \leq n$$

that is,

$$\sum_{Loc(x_i)=k} d_i x_i \leq \sum_{Loc(x_i)=k} d_i D(x_i)$$

Finally, if c_k *appears in an inequality of the form*

$$\sum_{Loc(x_i)=k} d_i x_i + c_k < n$$

We replace it with $-\sum_{Loc(x_i)=k} d_i D(x_i) + n - 1$, *which gives*

$$\sum_{Loc(x_i)=k} d_i x_i - \sum_{Loc(x_i)=k} d_i D(x_i) + n - 1 < n$$

or

$$\sum_{Loc(x_i)=k} d_i x_i - 1 < \sum_{Loc(x_i)=k} d_i D(x_i)$$

or equivalently

$$\sum_{Loc(x_i)=k} d_i x_i \leq \sum_{Loc(x_i)=k} d_i D(x_i)$$

which is the same expression as in the $\leq$ *case.*

We now show the above is a valid treaty configuration.

The first requirement is that the local treaties need to hold on D. This is straightforward because we know φ_Γ holds on D. Observing the formulas we gave above (with the appropriate expressions for c_k substituted in) makes it clear that all clauses of all local treaties must hold on D.

We now show that for any database D', satisfying all the local treaties implies satisfying the global treaties.

Suppose for a contradiction that all local treaties are satisfied on D' but φ_Γ does not hold on D'. Thus there must be at least one conjunct in φ_Γ that is false on D', although the corresponding conjunct is true in all the φ_{Γ_i} on D'. We argue by cases, depending on whether the conjunct is an equality or inequality.

Start with the equality case. We know that the equality holds on the original D, so

$$\sum_i d_i D(x_i) = n$$

Under the treaty configuration we have constructed, the local expression for site k will have the following form:

$$\sum_{x_i \in L, Loc(x_i)=k} d_i x_i + \sum_{x_j \in L, Loc(x_j) \neq k} d_j D(x_j) = n$$

By assumption this local expression is true at each site k. Summing over all sites yields the equality

$$\sum_i d_i D'(x_i) + (k-1)[\sum_i d_i D(x_i)] = kn$$

This is true because each variable is local to one site and remote to the $k-1$ *other ones.*

We can now use the fact that the expression is true on D *and subtract* $(k-1)[\sum_i d_i D(x_i)]$ *from both sides, yielding*

$$\sum_i d_i D'(x_i) = n$$

giving the desired contradiction.

In the inequality case (both $<$ *and* $\leq$ *as explained above), the local constraint that holds at site k is*

$$\sum_{Loc(x_i)=k} d_i D'(x_i) \leq \sum_{Loc(x_i)=k} d_i D(x_i)$$

Summing these over all k sites yields the following, as each variable appears exactly on one site:

$$\sum_i d_i D'(x_i) \leq \sum_i d_i D(x_i)$$

But by assumption the treaty holds on D so we know

$$\sum_i d_i D(x_i) \leq n$$

and by transitivity

$$\sum_i d_i D'(x_i) \leq n$$

giving the desired contradiction.

In general, many valid treaty configurations may exist. Our final step is to run an optimization to find a configuration that yields local treaties with a low probability of being violated. As mentioned, we assume we have a model that describes expected future workloads; the details of this model and how it is generated are independent of our algorithm. All that we need is a way to use the model to sample (generate) a number of possible future system executions. Once these are available, we create a MaxSAT instance and use a solver to generate the optimal treaty configuration for these future executions. The number and length of future executions we consider are tunable parameters that allow us to trade off quality of treaties versus time spent in the solver. The details of how we generate the MaxSAT instance are described next.

Algorithm 1 computes valid treaty configurations. It has two tunable parameters: a lookahead interval L and a cost factor f. We examine f possible future system executions, where each execution is a sequence of L transactions and is constructed using our workload model. We search for a configuration that minimizes the number of treaty violations in our f executions. For larger f and L, the probability of a future treaty violation decreases, but the search for an optimal configuration takes more time to run.

The configuration search is performed by a suitable MaxSAT solver. We assume that we have an efficient solver that takes as input a first-order logic for-

mula, finds the largest satisfiable subset of constraints that includes all the *hard constraints* and also produces a model (a concrete assignment of values to configuration variables) which satisfies this subset of constraints.

We now discuss the algorithm in more detail. The algorithm first generates θ_h—this is a formula that says the local treaties must enforce the global treaty. This must always be true for a valid treaty configuration, so we will have θ_h as a hard constraint (Line 4). Next (Lines 6-12) the algorithm generates f sequences of transactions of length L using the workload model. For each sequence, it simulates the execution of transactions in the sequence to produce a sequence of databases, one per transactional write (Line 8). The desired behavior is that none of the databases in the sequence violate the local treaties. We therefore add an appropriate constraint for each database in the sequence (Line 10). This process yields a set Θ_s of soft constraints, which are fed to the MaxSAT solver together with the hard constraints in θ_h (Line 13).

Algorithm 1 Finding a valid treaty configuration that reduces the probability of future violation.

1: Let L be the lookahead interval.
2: Let f be the cost factor.
3: Let $\varphi = \bigwedge_{i \in [1,K]} \varphi_{\Gamma_i}$ {Conjunction of local treaty templates.}
4: $\theta_h = \{\forall\ Vars(\varphi_\Gamma),\ \ \bigwedge_{\{i \in [1:K]\}} (\varphi_{\Gamma_i}) \Rightarrow \varphi_\Gamma\}$ {Local treaties enforce global treaty.}
5: $\Theta_s = \emptyset$
6: **for** $i = 1$ to f **do**
7: Generate a sequence S of L transactions using model
8: Let $[D_0, D_1, ..., D_L]$ be the sequence of databases obtained by executing S on D (we have $D_0 = D$).
9: **for** $j = 1$ to L **do**
10: $\Theta_s = \Theta_s \cup \{\varphi(D_j)\}$ {Treaty holds on D_j.}
11: **end for**
12: **end for**
13: $C = MaxSAT(\{\theta_h\} \cup \Theta_s)$ {Hard constraints θ_h, soft Θ_s}

We illustrate an execution of Algorithm 1 by continuing with our example

and our local treaty templates $\varphi_{\Gamma_1} := (x + c_y \geq 20)$ and $\varphi_{\Gamma_2} := (c_x + y \geq 20)$. We have $\theta_h = (\forall [x, y], \varphi_{\Gamma_1} \wedge \varphi_{\Gamma_2} \Rightarrow \varphi_\Gamma)$.

Suppose that $L = 3$ and $f = 3$ and transaction T_3 is twice as likely as T_4. We construct 3 possible transaction sequences of length L drawn from this distribution. Suppose these are $S_1 = [T_3; T_3; T_4]$, $S_2 = [T_3; T_3; T_3]$ and $S_3 = [T_3; T_4; T_3]$. If we execute S_1 on the initial database we obtain the sequence of states $[(10, 13); (9, 13); (8, 13); (8, 12)]$ where each element represents an ordered pair of x and y values. This sequence can be computed by applying the symbolic write trace for each transaction in S_1.

The desired behavior is that no database in the above sequence violates a local treaty. Thus $\varphi_{\Gamma_1} \wedge \varphi_{\Gamma_2}$ must hold on each of the four databases. Plugging in the x and y values from each database into $\varphi_{\Gamma_1} \wedge \varphi_{\Gamma_2}$, taking the conjunction and performing some simplification yields the soft constraints $\{(c_y \geq 12), (c_x \geq 8)\}$. Repeating the procedure for S_2 and S_3, we get soft constraints $\{(c_y \geq 13), (c_x \geq 7)\}$ and $\{(c_y \geq 12), (c_x \geq 8)\}$.

By passing all these constraints to a MaxSAT solver, we find that not all of the soft constraints can be satisfied in addition to θ_h. However, for $c_y = 12$ and $c_x = 8$, we can satisfy both the first and third set of soft constraints. The algorithm therefore chooses $C = \{(c_y = 12, c_x = 8)\}$. Note that this configuration allows more flexibility to the site at which T_3 runs, since it is more frequent. As desired, we have minimized the probability of a treaty violation subject to the computational constraints f and L.

3.3.4 Lifting assumption 3.3.1

We now explain how to handle transactions that violate Assumption 3.3.1. An example of such a transaction would be one that reads the value of remote object y and copies it into the local object x. Intuitively, if we wish to avoid the remote read of y yet guarantee correct execution, we need a global treaty that guarantees the value of y will never change. We can achieve this by adding a constraint to the local treaty *on the site where y resides*. To do this, we need to perform a post-processing step on the local treaty templates. Specifically, for every transaction T executing at site i, we consider the symbolic write trace ϕ_T associated with φ_Γ in the symbolic table. For every variable x referenced in ϕ_T such that $Loc(x) \neq i$, we substitute it with a new local variable c_x and modify $\varphi_{\Gamma_{Loc(x)}}$ to $\varphi_{\Gamma_{Loc(x)}} \wedge (x = c_x)$. Once this postprocessing step is complete, we can run Algorithm 1 as before.

3.3.5 Summary

At the core of the homeostasis protocol is the global treaty that must be maintained for correctness. We have explained how to generate the global treaty using symbolic tables, and how to factorize it into local treaties taking into account expected workload characteristics. With these crucial algorithms in place, we are now ready to proceed to a more in-depth overview of our system.

3.4 Homeostasis In Practice

In this section, we highlight some of the practical challenges in implementing the homeostasis protocol and discuss specific design decisions we made in our prototype implementation.

3.4.1 System design

Our system has two main components – an offline compiler which analyzes application transaction code, and an online server which receives transaction execution requests from clients and coordinates with other servers to establish and maintain treaties. We discuss each of these components next.

Offline preprocessing As shown in Figure 3.8, there are three components in offline preprocessing: the IR (Intermediate Representation) compiler, the analyzer and the protocol initializer.

The IR compiler takes as input the source code for all transactions which the application may issue. This code is compiled into our language $\mathcal{L}$. The complexity of the compilation process depends on the source language; it is very simple for transactions written in Transact-SQL using only `SELECT-FROM-WHERE` clauses and `IF-THEN-ELSE` conditionals.

The analyzer computes (joint) symbolic tables for the transactions in $\mathcal{L}$. In doing so, it applies a number of compression techniques to exploit independence properties and keep the size of the symbolic tables small.

Often transaction code operates on multiple database objects independently;

for example, the TPC-C New Order transaction orders several different items. The stock level of each item affects the transaction behavior, but each item affects a different portion of the code. Using a read-write dependency analysis like the one in SDD-1 [93], we identify such points of independence and use them to encode symbolic tables more concisely in a factorized manner.

Furthermore, the code that the analyzer receives is not arbitrary $\mathcal{L}$ code; it has been translated from a higher-level language and as such has a particular structure that the analyzer can exploit. We allow the IR compiler to give the analyzer "hints" about this structure to allow better compression. Finally, transactions may take integer parameters, and the behavior of the transaction obviously depends on the concrete parameter values. Rather than instantiate parameters now, we push the parameterization into the symbolic tables.

The last component is the protocol initializer. This sets up the *treaty table* – a data structure that at any given time contains the current global treaty and the current local treaty configuration. The treaty table is thus dependent on the current database state; it is initialized offline based on the database state before the system starts accepting transaction requests. Subsequently, it is updated at each treaty negotiation in the online component.

The protocol initializer also performs some further setup for the online component. For every symbolic write trace in the symbolic tables produced by the analyzer, it creates and registers a stored procedure which takes in the same parameters as the original transaction and "applies" the symbolic write trace. The stored procedure also includes checks for the satisfaction of the corresponding treaty as maintained in the treaty table. The stored procedure returns a boolean flag indicating whether the local treaty is violated after execution. The protocol

initializer also creates a catalog that maps transactions to corresponding stored procedures in the treaty table.

Online execution The online component accepts and executes transactions using the homeostasis protocol. When a transaction execution request arrives from the clients, the system identifies the appropriate stored procedure in the catalog created during offline preprocessing. The server executes the stored procedure within the scope of a transaction. If the local treaty associated with the stored procedure is satisfied, then the transaction commits locally. Otherwise, the server invokes the treaty negotiator to synchronize with other servers and renegotiate a set of treaties. The negotiator uses an optimizer such as a SAT solver to determine local treaties. It then updates the treaty table and propagates the new treaties to all the other nodes. Therefore, every treaty negotiation requires two rounds of global communication—one for synchronizing database state across nodes and one for communicating the new treaties.

3.4.2 Implementation details

Our system is implemented in Java as middleware built on top of the MySQL InnoDB Engine. Each system instance has a similar setup and communicates with the other instances through network channels. When handling failures, we currently rely on the recovery mechanisms of the underlying database. All in-memory state can be recomputed after failure recovery. In the offline component, we use ANTLR-4 to generate a parser for transactions in $\mathcal{L}$. For finding optimal treaty configurations, we use the Fu-Malik Max SAT procedure [37] in the Microsoft Z3 SMT solver [29].

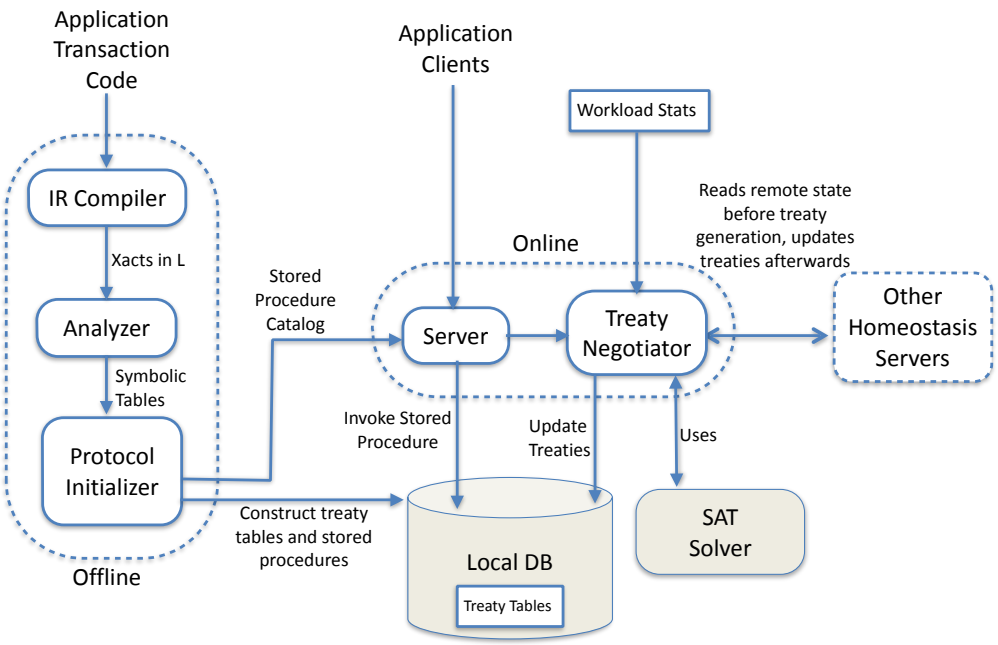

Figure 3.8: Homeostasis System Architecture.

3.5 Evaluation

We now show an experimental evaluation of a prototype implementation of the homeostasis protocol. We run a number of microbenchmarks (Section 3.5.1), as well as a set of experiments based on TPC-C [3](Section 3.5.2) to evaluate our implementation in a realistic setting. All experiments run in a replicated system.

3.5.1 Microbenchmarks

With our microbenchmark, we wanted to understand how the homeostasis protocol behaves in our intended use case – an OLTP system where treaty violation and negotiation are infrequent. In particular, we are interested in the following questions:

- Since our protocol reduces communication, will it yield more performance benefits as the network round trip time (RTT) between replicas increases?
- As the number of replicas increases, treaty negotiations will be more frequent, because each replica must be assigned a "smaller" treaty that is more likely to be violated. How does the performance change with the degree of replication?
- Each replica/server runs multiple clients that issue transactions. How does the performance change when the number of clients per server (i.e., the degree of concurrency) increases?
- How much benefit can we gain from the protocol as compared to the two obvious baselines — two-phase commit (2PC), and running all transactions locally without synchronizing?

To answer these questions, we designed a configurable workload inspired by an e-commerce application. We use a database with a single table `Stock` with just two attributes: item ID (`itemid` INT) and quantity (`qty` INT). The item ID is the primary key. The workload consists of a single parameterized transaction. It reads an item by (`itemid`), and updates the quantity as though placing an order. If the quantity is more than one, it decreases the quantity; otherwise, it refills it. The transaction template is shown below.

Listing 3.1: The microbenchmark transaction; @itemid is an input parameter, while REFILL is a constant.

```
SELECT qty FROM stock WHERE itemid=@itemid;
if (qty>1) then
    new_qty=qty-1
else
    new_qty=REFILL-1
UPDATE stock SET qty=new_qty WHERE itemid=@itemid;
```

We implemented two baseline transaction execution solutions: *local* and *two phase commit (2PC)*. In *local* mode, each replica executes the transactions locally without any communication; thus, database consistency across replicas is not guaranteed. Local mode provides a bare-bones performance baseline for how fast our transactions run locally. In 2PC mode, each transaction requires two rounds of communication. In the prepare phase, the source replica issues prepare requests to all replicas for execution of a particular transaction. If all replicas respond positively to the prepare request, then in the commit phase, the source replica issues a commit request. Otherwise, the transaction is aborted. This protocol ensures the database is consistent and all transactions run in full isolation. It provides a baseline of the performance of a geo-replicated system implemented in a classical way.

Our workload has several configurable parameters: network RTT, number of replicas (N_r), and number of clients per replica (N_c). By default, we set RTT to 100ms, the number of replicas to 2, and the number of clients per replica to 1. The database is populated with 10K items, and the value for REFILL is set to 100.

All the experiments are run on a single Amazon EC2 c3.8xlarge instance, with 32 vCPUs, 60GB memory, and 2x320GB SSDs, running Ubuntu 14.04 and MySQL Version 5.5.38 as the local database system. We run all replicas on the same instance, and we simulate different RTTs. For each run, we start the system for 10 seconds as a warm-up phase, and then measure the performance for the next 120 seconds. All data points are averages over three runs; the deviation between runs is insignificant and is therefore omitted from the figures for clarity.

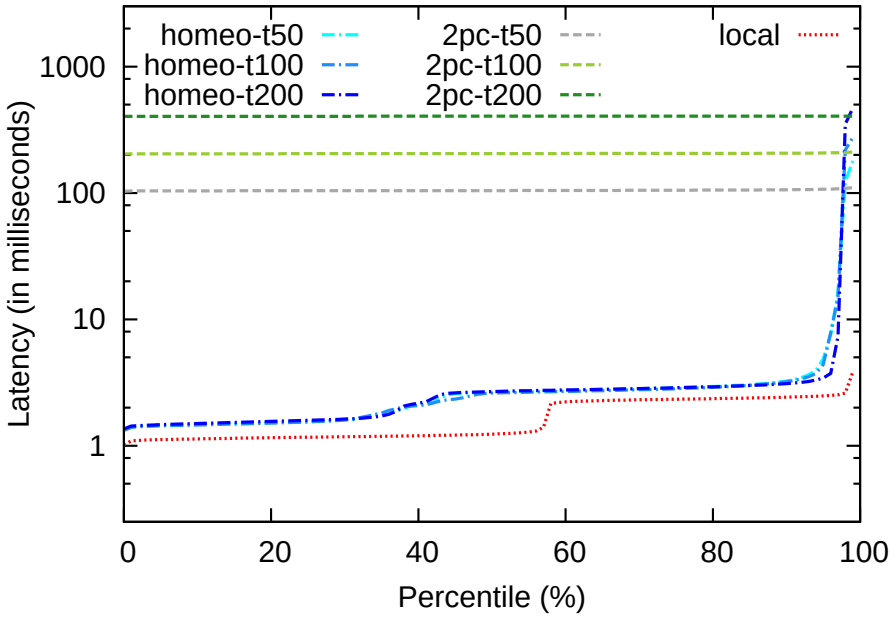

Figure 3.9: Latency with network RTT ($N_r = 2, N_c = 1$)

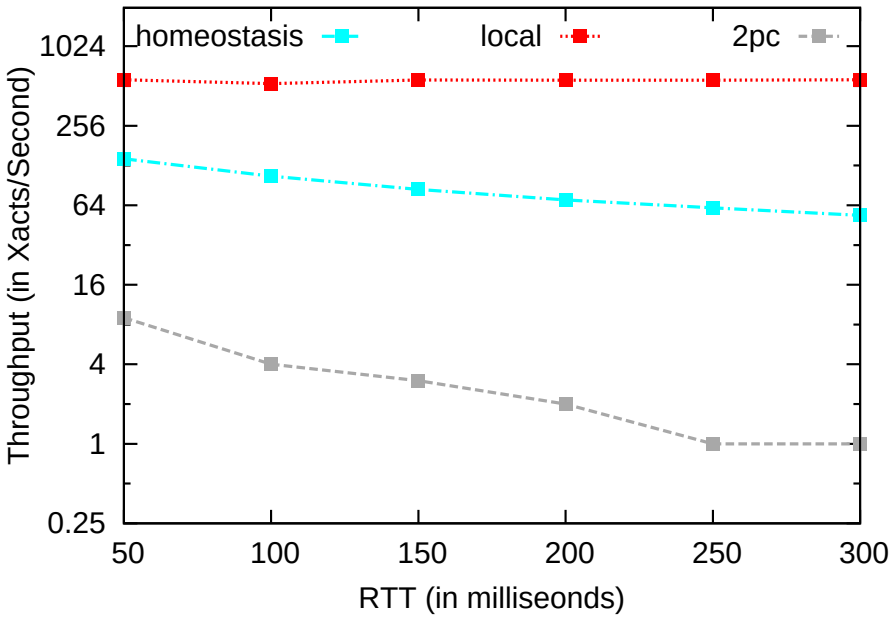

Figure 3.10: Throughput with network RTT ($N_r = 2, N_c = 1$)

Varying RTT Our first experiment varies the network RTT from 50ms to 300ms, using the default values for all other parameters. Figure 3.9 shows the transaction latency by percentile. When using the homeostasis protocol, 98% of the transactions execute locally, with latency less than 4ms. When a transaction

requires treaty negotiation, the latency goes up to around 2RTT. In comparison, in the local mode, all the transactions execute locally and complete in about 2ms. This additional overhead of less than 1ms of latency per transaction can be attributed to the time required for checking local treaties. For transactions which cause a local treaty violation, in addition to the 2RTT communication overhead, an additional overhead of less than 50ms is incurred in finding new treaties using the solver. This treaty computation overhead is a tunable parameter and can be reduced, though at the cost of more frequent treaty violations. Under 2PC, each transaction requires two RTTs, and thus the transaction latency is consistently twice the RTT. Figure 3.10 shows the throughput per second for each replica. In 2PC mode, it is less than 10 transactions per second due to the network communication cost. The homeostasis protocol allows 16x-60x more throughput than 2PC, depending on the RTT setting. The difference between the throughput for the homeostasis protocol and local mode can be attributed to the small fraction of transactions which require synchronization; for example, if only 2% of transactions require treaty negotiation and the RTT is 100ms, this leads to an average latency of 4*0.98+200*0.02=7.92ms. With $N_c = 1$, this represents a halving of the throughput as compared to local mode.

Varying number of replicas Next, we vary the number of replicas from 2 to 5, while setting the other parameters to their default values. Figure 3.11 shows the transaction latency profile. With a higher number of replicas, the local treaties are expected to be more conservative and therefore lead to more frequent violations; this leads to an increase in latency. The transaction latency also increases for the local and 2PC modes. In the local case, this is due to the increased resource contention since our experimental setup requires us to run all the replicas on the same server. In 2PC, each transaction stays in the system

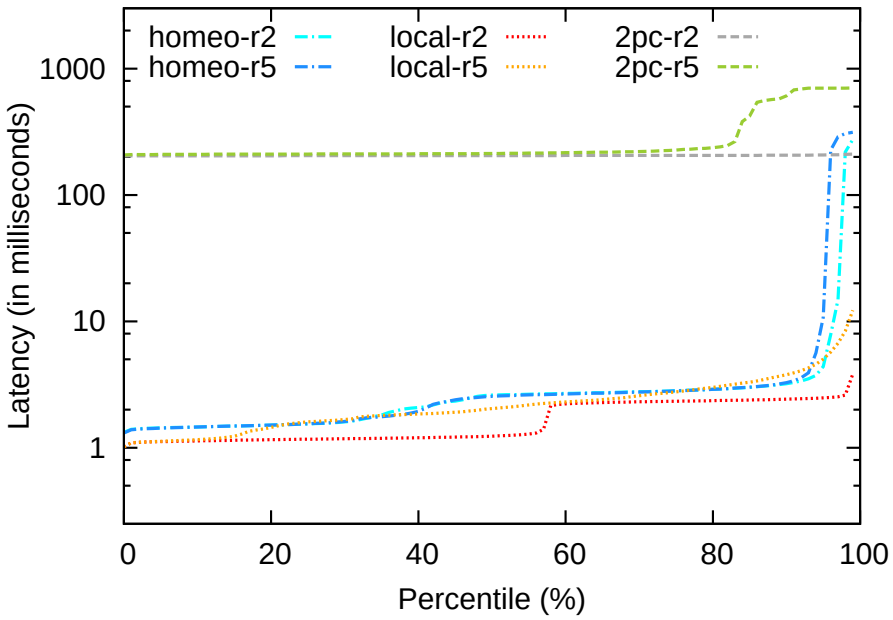

Figure 3.11: Latency with the number of replicas ($RTT = 100ms, N_c = 1$)

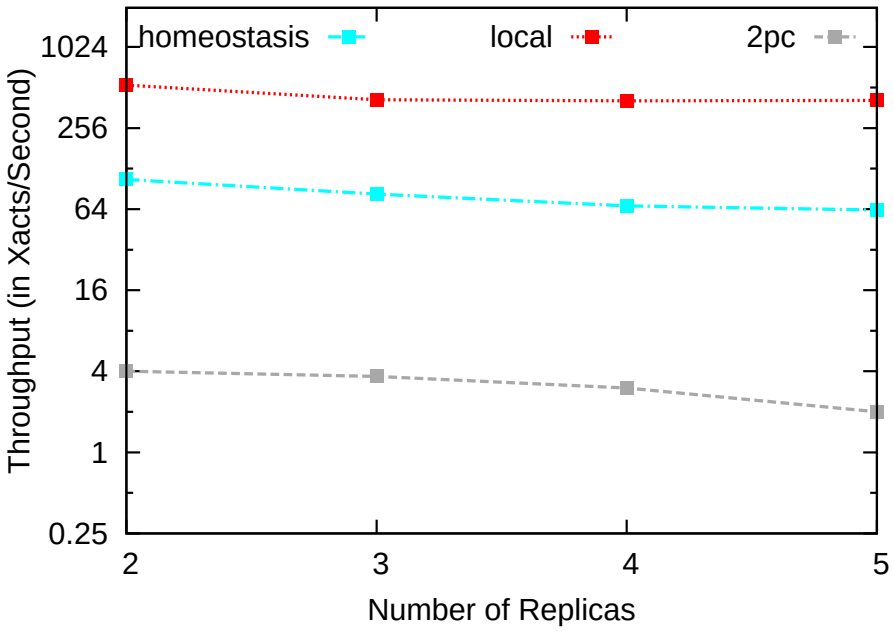

Figure 3.12: Throughput with the number of replicas ($RTT = 100ms, N_c = 1$)

longer since it has to wait for communication with more replicas; this causes an increase in conflict rates which further increases transaction latency. Figure 3.12 shows the throughput per second for each replica. As expected, the throughput decreases for all modes as the degree of replication increases.

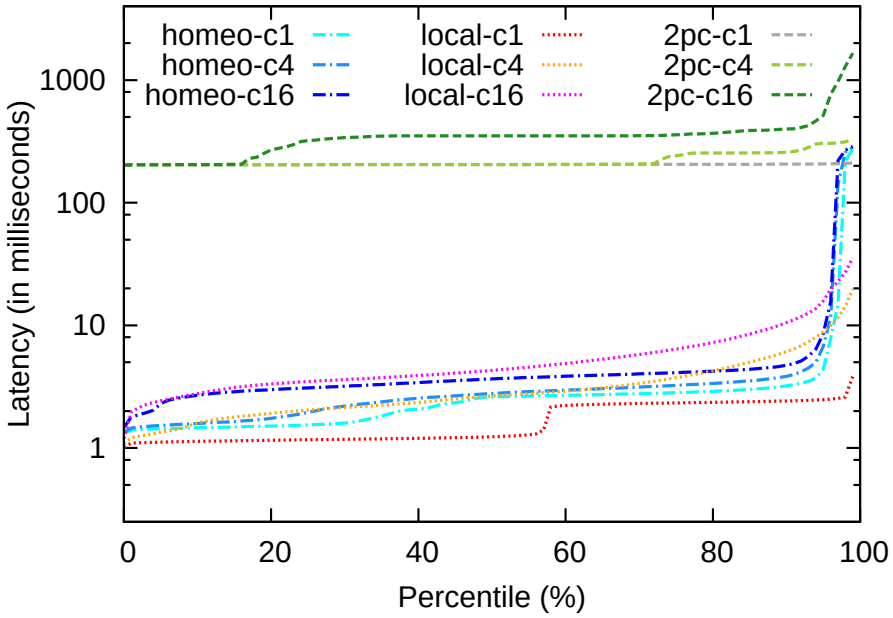

Figure 3.13: Latency with the number of clients ($N_r = 2, RTT = 100ms$)

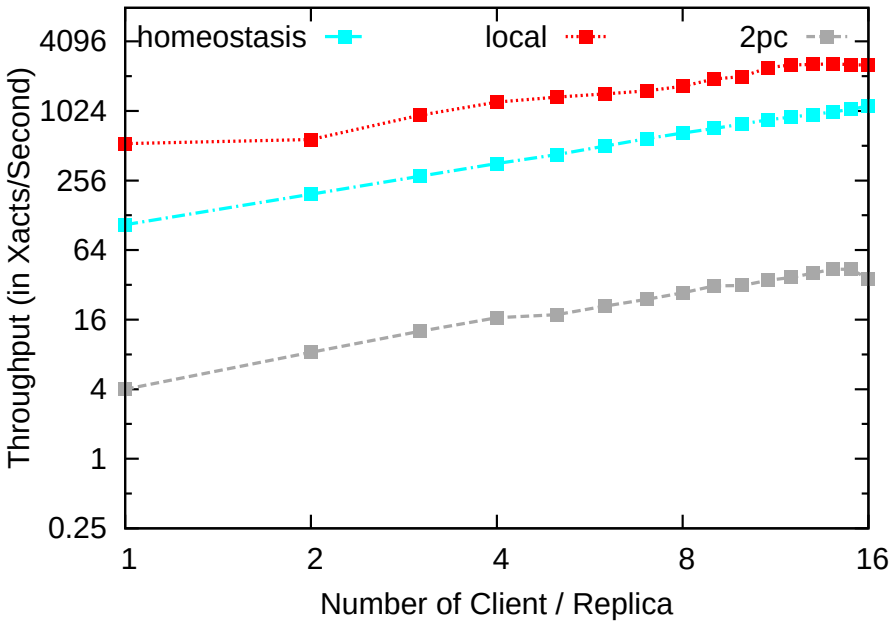

Figure 3.14: Throughput with the number of clients ($N_r = 2, RTT = 100ms$)

Varying number of clients Finally, we vary the number of clients per replica from 1 to 16, while setting the other parameters to their default values. Figure 3.13 shows the transaction latency profile. In all modes, the transaction latency increases with the number of clients due to higher data and resource con-

tention. This is analogous to what we observed in the previous experiment. The curves for the homeostasis protocol clearly show the latency difference between the transactions that execute locally and those that require communication. Figure 3.14 shows the throughput per replica. When using the homeostasis protocol with 16 clients, the throughput per client reaches 80% of the throughput per client we observe for 4 clients, indicating good scalability with the number of clients.

3.5.2 TPC-C

To evaluate the performance of the homeostasis protocol over more realistic workloads with multiple transactions, larger databases, and non-uniform workload characteristics, we created a set of experiments based on the TPC-C benchmark.

Data The benchmark describes an order-entry and fulfillment environment. The database contains the following tables: `Warehouse`, `District`, `Orders`, `NewOrder`, `Customers`, `Items`, `Stock` and `Orderline` with attributes as specified in the TPC-C benchmark. The initial database is populated with 10000 customers, 10 warehouses, 10 districts per warehouse and 1000 items per district for a total of 100,000 entries in the `Stock` table. Initial stock levels are set to a random value between 0 and 100.

Workload We use three transactions based on the three most frequent transactions in TPC-C. The `New Order` transaction places a new order for some quantity (chosen uniformly at random between 1 to 5) of a particular item from a particular district and at a particular warehouse. The `Payment` transaction

updates the customer, district and warehouse balances as though processing a payment, and the `Delivery` transaction fulfills the oldest order at a particular warehouse and district.

The `New Order` and `Delivery` transaction both operate on the `New Order` table, as the former inserts new tuples into it while the latter deletes the oldest tuple. However, as long as the `New Order` table is not empty, instances of these two transactions do not need to synchronize with each other. Our analyzer, with minimal hints, is able to set up suitable treaties for this situation. Note that instances of the `Delivery` transactions do need to synchronize *with each other* to ensure that the globally oldest order is deleted each time.

For all experiments, we issue a mix of 45% `New Order`, 45% `Payment` and 10% `Delivery` transactions. In order to simulate a skew in the workload, we mark 1% of the items as "hot" and vary the percentage of `New Order` transactions that order hot items. We denote this percentage as H. For example, a value of $H = 10$ indicates that 10% of all `New Order` transactions order the 1% hot items.

Setup We run all our experiments on c3.4xlarge Amazon EC2 instances (16 cores, 30GB memory, 2x160GB SSDs) deployed at the Virginia (UE), Oregon (UW), Ireland (IE), Singapore (SG) and Sao Paolo (BR) datacenters. The average round trip latencies between these datacenters are shown in Table 3.1. For all the experiments, we use a single c3.4xlarge node per datacenter. All two-replica experiments use instances from the UE and UW datacenters. There are eight clients per replica issuing transactions. All measurements are performed over a period of 500s after a warmup period of 100s. We only report measurements for the `New Order` transactions, following the TPC-C specification. For

comparison, we run the same workload against an implementation of the two-phase commit protocol. All reported values are averages of at least three runs with a variance of less than 6% in all experiments.

	UE	UW	IE	SG	BR
UE	< 1	64	80	243	164
UW	-	< 1	170	210	227
IE	-	-	< 1	285	235
SG	-	-	-	< 1	372
BR	-	-	-	-	< 1

Table 3.1: Average RTTs between Amazon datacenters (in milliseconds)

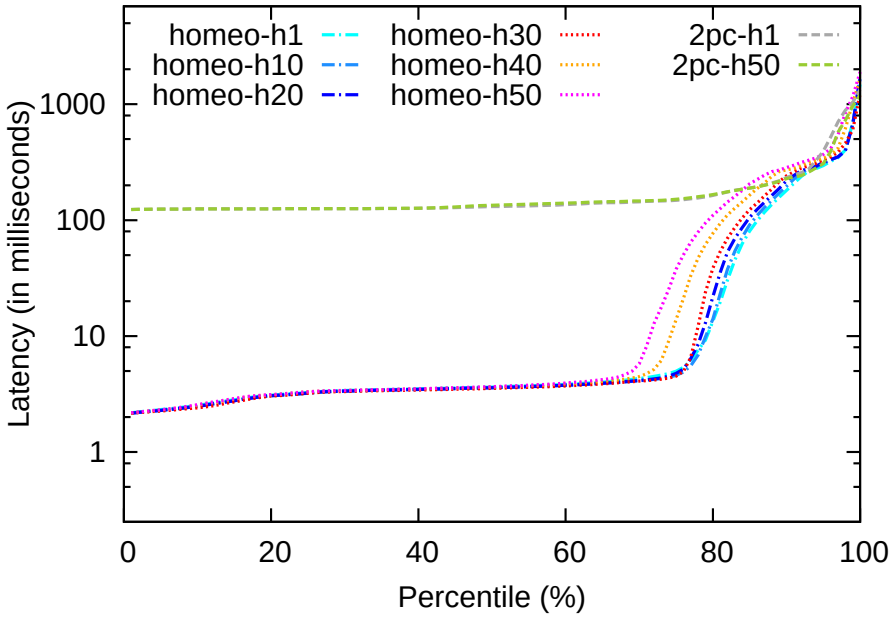

Figure 3.15: Latency with workload skew($N_r = 2, N_c = 8$)

Varying Workload Skew For this experiment we vary H from 1 to 50, i.e., the percentage of transactions that involve hot items varies from 1% to 50%. The latency profile for different values of H is shown in Figure 3.15. As the value of H increases, the treaties for the hot items are violated more often, so a higher fraction of transactions takes a latency hit. This causes the rightward shift of the latency profiles. In comparison, the latency profile for two-phase commit (2PC) is relatively unaffected as it always incurs a two RTT latency hit.

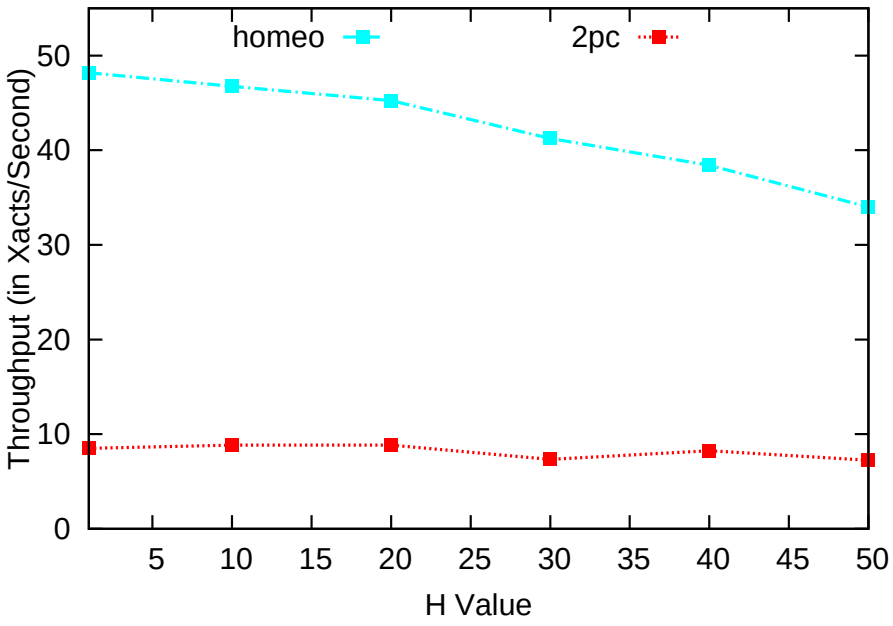

Figure 3.16: Throughput with workload skew($N_r = 2, N_c = 8$)

As shown in Figure 3.16, the throughput for 2PC drops with increased H due to an increased rate of conflicts. The throughput for the homeostasis protocol drops as well, but the throughput per replica is still significantly higher than that for 2PC. Note that we only show throughput numbers for the `New Order` transaction which constitutes 45% of the workload. The actual number of successful transactions committed by the system per second is more than twice this value. We can increase throughput by running more clients per replica, as we did for the microbenchmark experiments; we omit these results due to space constraints.

Varying the number of replicas For this experiment, we set the value of H to 10 and measure the latency and throughput of the `New Order` transactions as we increase the number of replicas. The replicas are added in the order UE, UW, IE, SG and BR. The latency profile and the throughput per replica are shown in Figures 3.17 and 3.18. As we add replicas, the maximum RTT between any two

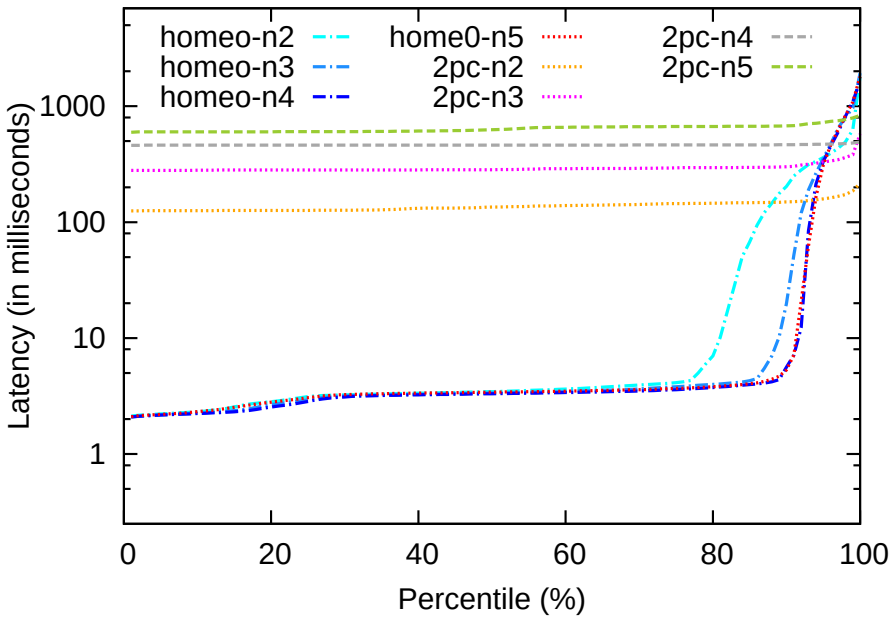

Figure 3.17: Latency with the number of replicas($N_c = 8, H = 10$)

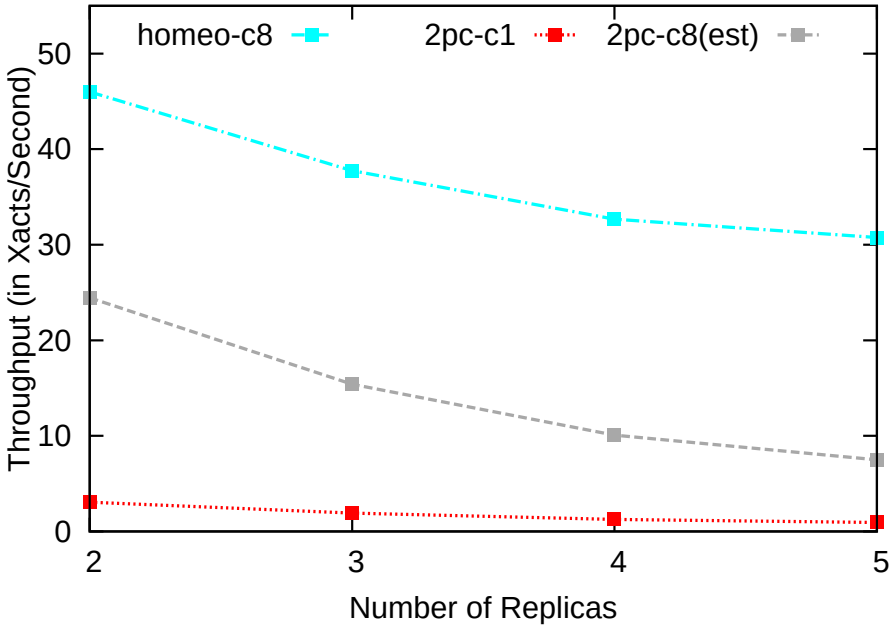

Figure 3.18: Throughput with the number of replicas($H = 10$)

replicas increases. This manifests itself towards the 98^{th} percentile as an upward shift in the latency profiles. With five replicas, the treaties become more expensive to compute, which also contributes to the upwards latency shift. On the other hand, with fewer replicas, the throughput is significantly higher which

means that with more replica, a higher fraction of transactions causes a treaty violation. This explains the leftward shift of the inflection point on the curves as the number of replicas decreases. In all cases, the `New Order` throughput values for the homeostasis protocol are substantially higher as compared to the 2PC baseline. In our 2PC implementation, we only use a single client per replica since with a larger number of clients, conflicts caused frequent transaction aborts. Thus as a very conservative upper bound, Figure 3.18 also shows an estimated maximum throughput for 2PC obtained by multiplying the measured throughput (with a single client) by a factor of 8 to give a conservative upper bound for 2PC; as can be seen from the figure, this estimate still has a significant lower throughput than the homeostatis protocol. Thus the homeostatis protocol clearly outperforms 2PC in all situations.

3.6 Summary

In this chapter, we introduced a new mechanism for implementing transactions that guarantees correct execution in a multi-node system while minimizing the communication needed to ensure consistency. Our key insight is to extract consistency requirements from transaction code *automatically via program analysis* rather than requiring human input, which enables a fully automated adaptive consistency management protocol. We have presented the method as a general framework and provided concrete implementations for each component in the framework.

CHAPTER 4

RELATED WORK

If I have seen further it is by standing on the shoulders of Giants.

— Isaac Newton

This dissertation relies on decades of research across multiple fields of computer science including database systems, systems, programming languages and constraint satisfaction. In this chapter, we briefly overview related work from each of these communities organized by the manner in which they relate to the work presented in this dissertation.

4.1 Possible World Representations

A quantum state of the database can be viewed as a set of possible database states which satisfy certain constraints induced by the committed transactions. This set can have a cardinality exponential in the number of pending transactions, and a naive extensional representation of all such states is computationally infeasible. In this section, we summarize work related to possible representations of Quantum Databases.

Our current implementation of Quantum Database models the certain part of the database extensionally as normal database tables, and the uncertain part

of the database as a constraint satisfaction problem and a set of pending updates. The set of updates when applied to any feasible solution to the constraint satisfaction problem produce a possible database state.

Quantum databases share the problem of representing possible worlds with probabilistic databases [99]. The power of a representation system for a probabilistic database is closely tied to the complexity of query processing – the more expressive the representation, the greater is the complexity of processing a query. Some popular representations of probabilistic databases include c-tables [54], pc-tables [44], World-Set-Decompositions [8], and U-relations [7]. While such representation systems can be used to represent quantum database, determining the representation which meets the requirements of quantum databases closely is future work.

A conceptual difference of Quantum Database with probabilistic database is the fact that uncertainty is completely internal to the database. In other words, while queries over probabilistic databases follow either the possible answer set semantics or the possible answer semantics, queries over quantum database are always deterministic. Such determinism is achieved by selectively eliminating uncertainties within the database before processing the query. The query is always executed on the deterministic relational part of the quantum state.

More generally, the possible worlds semantics plays an important role in certain modal logics, notably logics of knowledge [33].

Constraint databases [89, 63] have some conceptual similarities to quantum databases, particularly since both classes of systems work with intensional representations of state and/or relations. However, quantum databases do not

share the constraint database goal of representing and reasoning about relations that hold over an infinite universe. Conversely, work on constraint databases does not focus on representing and maintaining a set of possible worlds where the possibilities evolve as the database is modified.

4.2 Incremental Constraint Satisfaction

A core aspect of maintaining a quantum database is checking and maintaining the satisfiability of the composed transaction formula. Implementing the satisfiability check in the naïve way with relational queries is sub-optimal in the general case, as the resulting query has a large number of joins and is not well-suited to the capabilities of a traditional relational optimizer. We can rely on existing work in two domains to overcome this limitation.

First, this problem is an instance of the Satisfiability problem, which is known to have phase transitions [95]. Most real-world resource allocation problems are under-constrained at least in the initial phase, e.g., when seats on a given flight first go on sale. As resources are allocated the problem tends to reach a critical ratio of degrees of freedom (variables) vs. constraints at which the problem is hard; both comfortably under- and over-constrained problems tend to be easy to solve [95]. By identifying the difficulty of checking satisfiability, a quantum database can switch to a more aggressive fixing phase favoring faster response times over better assignments.

Second, there has been some work in integrating databases with constraint solvers or satisfiability checkers. We discuss four such existing systems and identify opportunities for a deeper integration of data processing capabilities of

databases with state-of-the-art solvers.

Tiresias [74] is a system that supports *how to* queries. How-to queries are queries of the form: "How should the input change in order to achieve the desired output". The system converts how-to queries specified over a relational database in a datalog-like language, TiQL, into a mixed integer program. The solution to the integer program is a *possible world* – a feasible database instance – which minimizes some objective specified in the how-to query. However, the mixed integer program created by Tiresias is linear in the size of the database. While such an expensive conversion is acceptable for how-to analytics queries, in quantum database the invariant check needs to be performed frequently as transactions execute.Quantum database exploits previously found solutions by trying to extend them instead of re-solving the constraint satisfaction problem from scratch each time. The probability of such extensibility is typically high since the changes made to the database by the transactions are usually marginal.

COnstraint LOGic EngiNE (Cologne) [68] is a declarative optimization platform that enables constraint optimization problems to be declaratively specified in a datalog-like language, Colog, and incrementally executed in distributed systems. Colog programs have datalog rules for derivations and constraint rules for specifying constraints. Cologne is implemented by integrating a distributed query processor used in declarative networking with an off-the-shelf constraint solver. The problem of maintaining the invariant in Quantum Database could indeed be mapped to a program in Colog. However, as new transactions get executed over Quantum Database, the Colog program representing the invariant would also change. While Cologne can do incremental optimization as the data changes, it does not support incremental program modifications. How-

ever, such scenarios do not arise naturally in the targeted application domain for Cologne.

DLV [66] is generally recognized to be the state-of-the-art implementation of disjunctive logic programs. Disjunctive datalog is an extension of datalog in which the logical OR expression is allowed to appear in the rules. Insertions of resources in quantum database could result in invariants with disjunctions. For quantum database workloads in which resource deletions and insertions are frequent, the quantum database invariants can have a number of disjuncts. Satisfiability check for such instances can potentially be done more efficiently using DLV. Our initial implementation adopts a naïve approach and processes disjuncts separately after converting each disjunct to an independent conjunctive query.

Finally, we can leverage state-of-the-art Satisfiability Modulo Theory (SMT) solvers [17, 30], which are essentially SAT solvers where the interpretation of some symbols is constrained by a background theory. Identifying the appropriate background theory for quantum databases and designing an algorithm which splits the task of satisfiability checking between the database and the solver to achieve maximum efficiency requires addressing many research problems, both theoretical and systems-related. The μZ tool [53] is a promising research direction which integrates the state-of-the-art SMT solver Z3 with a bottom-up Datalog engine for fixed point computations with constraints. μZ provides a pluggable and composable API for adding alternative finite table implementations of relational algebra operations. The potential integration of μZ with a standard relational database engine like MySQL could provide an ideal engine for Quantum databases.

4.3 Weaker Notions of Serializability

Since quantum databases post-pones the execution of certain updates in the transactions beyond the transaction commits, histories produced by the Quantum database are not serializable in the traditional sense. In this section, we first recap the traditional notions of serializability and then discuss weaker notions of serializability which are closely related to the notion of semantic serializability.

Serializability has been accepted as a strong correctness criterion for concurrent execution of transactions. A transaction schedule with interleaved operations of multiple transactions is considered to be serializable if it is equivalent to some serial schedule for the same set of transactions. This is the highest level of transaction isolation. However, enforcing strict serializability reduces concurrency, and therefore, the ANSI/ISO SQL-92 specifications [4] defined four levels of isolation which trade-off correctness for concurrency. In decreasing order of concurrency, they are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ and SERIALIZABLE. Berenson et al. [15] defined each of these levels more formally, and introduced Snapshot Isolation as an isolation level in between READ COMMITTED and REPEATABLE READ. However, these definitions were overly restrictive and specific to locking implementations. Adya et al. [5] proposed generalized definitions of isolation levels that not only apply to locking implementations, but also to optimistic and multi-version concurrency control schemes. In certain environments, where potential for conflicting operations is low and locking is expensive, such as large scale wide-area distributed systems, optimistic mechanisms [62] are the scheme of choice.

The above notions of serializability, and the concurrency control protocols implementing them do not rely on the semantics of the transactions. For certain set of applications, non-serializable schedules which do not violate application specific consistency requirements can be permitted. We next discuss approaches which relax notions of serializability, either for improving performance by added concurrency, or for supporting new abstractions.

The *intent* of a transaction is a high-level semantic description of the transformation induced by the transaction on the data state. Transactional intent [36] can be mapped to a sequence of operations with well defined semantics, each of which transforms the database state. A log of transactional intents can be replayed, if required, to deterministically produce a consistent database state. For example, in a calendar system using intent, a meeting intended to be scheduled after an architectural review meeting, and before Friday can be rescheduled automatically based on the intent. Quantum database precisely captures such intent in the specific domain of acquisition of resources and delays the conversion of such intent into concrete state unless forced by the application or by a read. Moreover, quantum database composes intents of new transactions over committed intents, and does so efficiently by exploiting the restriction of intent to be over resources.

4.3.1 Long-lived Business Transactions

Business transactions as opposed to OLTP style database transactions typically take a relatively longer period of time for execution, even without interference from concurrent transactions. The longer duration of such transactions can be

due to lengthy computations in the transactions over many database objects, pauses for inputs from users, or the overhead of coordinating across multiple (often opaque) autonomous services glued together in a loosely-coupled virtual application. Examples of such transactions include producing monthly account statements at a bank, transactions to collect statistics over an entire database or, a transaction to reserve an airline, hotel and car for a trip. The traditional locking mechanisms enforcing ACID guarantees do not scale well when applied to such longer running transactions.

Hector Molina proposed *sagas* [41, 40, 39] as a relatively simple concept to improve performance in long-running transactions. Sagas are long-lived transactions which can be written as a sequence of transactions that can be interleaved with other transactions. A system supporting sagas would guarantee that either all the transactions in a saga are completed successfully, or compensating transactions are run to amend the partial execution. Instead of compensation transactions, Quantum database supports optional conditions which are satisfied on a best-effort basis. While there is no direct analogy between sagas and resource transactions, a saga in which some nested transactions are optional is similar to a resource transaction with optional conditions. Sagas are closely related to nested transactions model [72, 76] that permits transactions to be nested, and requires serializability of transactions at every level, including the top level. Split-transactions [83] were proposed to deal with the uncertain duration, uncertain developments or interaction with other concurrent activities exhibited in open-ended activities by permitting transactions to commit data by splitting the transaction and committing one of the siblings thus formed.

The Escrow transactional method [80] permitted record updates by long-

lived transactions without requiring exclusive access to the modified records. The escrow method requires the database system to be an "expert" about certain types of transactional updates performed. While extendable to other updates, most commonly such updates involve incremental changes to aggregate quantities. This idea of exploiting incremental changes to aggregate quantities was separately proposed by Gawlick et al. [42] and Reuter [87]. Both these approaches dealt with the problem of "hot spots" in some applications. Hot spots were quantities which were very frequently updated – incremented or decremented – so that exclusive access to such quantities even for short durations could result in major bottlenecks in the system. By restricting access to such aggregate quantities through TEST and MODIFY requests, these approaches could was able to avoid exact reads and instead deal with the range in which this aggregate quantities fall in. Similar to quantum database, a limited uncertainty is deliberately introduced in the system to enable concurrent updates. The escrow transactional model designated such quantities to be of "Escrow Type", and similar to quantum database admits updates to escrow type only if it is not in conflict with previous escrow updates. Each such escrow update is registered in an escrow journal which are basically operational transformations of the escrow quantity. When a long-lived transaction finally commits or aborts, the associated escrow journal is applied as an update or discarded, as appropriate. The escrow transactional method also alludes to the possibility of exploiting the escrow model for better resource allocation by supporting "don't care" requests similar to choice in resource transactions of quantum database.

The quantum database middle-tier extracts out the resource request along with the associated constraints from a resource transaction specified in extended SQL. Such constraints which are maintained as part of the invariant after the

transaction commits, can be regarded as promises made by the system to the transaction. While a quantum database maintains these promises beyond transaction commits for better resource allocation, Greenfield et al. [45, 22, 55] used a similar technique to improve concurrency in long-running business transactions for service-based applications. They introduced *Promises* as an abstract way for a client to specify the resources they need to ensure that they can complete successfully. A granted promise guarantees that the request resources will be available when needed by later actions, but does not necessarily guarantee that any particular instance of the resource will be used to meet this promise. While promises have an associated expiry time, constraints in quantum database are resolved transparently as and when required, i.e., on reads affected or when triggered by the application. Quantum databases can be generalized to support the more general abstraction of promises. This of course would require more sophisticated reasoning and satisfiability checks for maintaining the invariant and ensuring all promises are met. A predecessor of Promises was the early ConTract [88] work. The authors introduced preconditions needed to allow actions within a workflow to execute successfully, and identified different styles of ensuring that these preconditions still hold at the time when applications rely on them in an execution. More recently, Helland and Haderle proposed *engagements* [50] as an abstraction for business transactions comprising many individual "classic" transactions spread across many disparate systems. Their framework relies on the promise protocol to implement eventually ACiD business transactions.

An alternative to using Quantum databases for an application is to rely on protocols such as the Tentative Hold Protocol [91, 90]. The Tentative Hold Protocol is an open, loosely coupled, messaging-based framework for exchange of

information between businesses prior to the actual transaction itself. The protocol defines an architecture that allows tentative, non-blocking holds or reservations to be requested for a business resource. We believe Quantum database adopts a principled approach to the same problem at the database level, and generalizes beyond simple holds on data-items.

4.3.2 Cooperating Transactions

While isolation has been hailed as a virtue for transaction processing system, in the era of Web 2.0, coordination among users to achieve complex data-driven tasks requires relaxing, if not abandoning, the traditional notions of isolation. The idea for allowing interdependence and coordination between transactions was first introduced in [57]. Subsequently, we proposed entangled queries [46], a mechanism that admits a limited form of interaction between database queries by automatically coordinating not on events or conditions, but on the choice of common values between the queries. Similar to entangled queries, enmeshed queries [21] are also an abstraction for declarative social coordination with different expressivity and algorithms which scale to larger coordination groups under certain limitations. Most real-world data management applications that involve coordination require not just queries, but a transaction-like abstraction that covers larger units of work. Entangled transactions [47] are transactions which have entangled queries embedded in them, and therefore cannot be executed in complete isolation. The paper proposed *oracle serializability* as a weaker notion of serializability which allows information flow between entangled transactions but only as much is necessary for successfully answering the entangled queries.

Entangled resource transactions are different from the (pure) entangled transactions [47] where coordination was required for successful execution. The execution model for entangled transactions does not allow an entangled transaction to commit until its partner(s) is also in the system; this means that entangled transactions must be executed in batches. Our new quantum database model allows entangled resource transactions to execute and commit individually. The benefit of using Quantum database as a platform for executing entangled transactions is that while Mickey is no longer guaranteed to sit next to Goofy, he is now guaranteed to have a seat for himself regardless of when and whether Goofy might submit his transaction.

There have been a few other attempts in the past to relax the requirement of atomic, serializable transactions to better support cooperative applications. Cooperative transaction hierarchies [77] allows the correctness specification for groups of designers to be tailored to the needs of the application by using patterns and conflicts to specify constraints imposed on a group's history. Nested transactions [72, 76] and Split-transactions [83] also allow some form of communication between the inner transactions.

4.4 Transaction Analysis

Exploiting application semantics to improve performance in concurrent settings has been explored extensively, from thirty years ago [38] up to today [9]. We refer the reader to [104] for a more complete survey, and only highlight the approaches most similar to ours.

Hector Molina proposed the use of application semantics to allow non-

serializable, yet semantically correct, schedules [38]. To produce such schedules, he proposed a mechanism which allows users to exploit their semantic knowledge by dividing transactions into steps, compatibility sets, and countersteps. The major drawback of this approach is that it requires the programmers to analyze the transactions and to write countersteps, both of which can be non-trivial. The Calvin [101] and Lynx [109] systems also perform static analysis of transactions to improve performance; however, they ensure that all transactions operate on consistent data, whereas we allow them to see inconsistent data if the inconsistency is "irrelevant".

There is extensive research on identifying cases where transactions or operations commute and allowing interleaving which are non-serializable yet correct [61, 67, 51, 96, 6]. Our solution extends this work, as the use of LR-slices allows transactions to commute in special cases where they might not commute in general.

Program analysis techniques from Status Quo [24, 26], Sloth [25] and those presented specifically in the context of lazy evaluation [35] can be used to further extract more expressive treaties by performing analysis of transactions written in higher level languages.

There is more general distributed systems work that explains how to detect whether global properties hold [20, 73], and – closer to our application setting – how to enforce them using local assertions [19]. However, this work does not address the question of how the local assertions may be computed.

Symbolic tables bear similarities to constructs from Owicki-Gries proofs [81] used in reasoning about correctness of parallel programs; however, we are ap-

plying them in a very different setting.

There is a large body of work in programming languages on using synthesis to develop concurrent programs [23, 103, 97, 102, 49]. Our work lies in a different problem space, as our applications are database transactions rather than general-purpose code, and our domain comes with extra challenges such as the need to optimize based on expected future workload.

4.5 Divergence Control Techniques

The idea of exploiting application semantics to resolve conflicting updates or divergent replicas has been explored in the past. In this section, we discuss some such divergence control techniques, and compare them to the Homeostasis protocol which at the core is a distributed divergence control protocol.

The demarcation protocol [14] was proposed as a technique for maintaining constraints in a distributed database systems. It allows individual sites to maintain high autonomy by identifying safe limits from explicit consistency constraints. The paper mainly focuses on constraints expressed as linear arithmetic inequalities, and approximate arithmetic equalities. The demarcation protocol defines limits for each variable in the inequality, and allows nodes to update the variable if the change is within limits. Limits associated with each variable are changed as required. The decision when to change the limits and how to manage the limits are policy decisions with multiple choices. Our work differs from demarcation protocol as we extract out the consistency requirements of the application from the transactions automatically, and therefore users do not have to explicitly specify the set of constraints which are to be enforced. Secondly, we

generalize beyond simple arithmetic inequalities. Finally, we adapt to changes in the workload by optimizing treaty factorization based on the expected transaction workload.

Epsilon serializability [85] is a correctness criterion which allows temporary and bounded inconsistency to be seen by queries. Epsilon serializability allows for more concurrency by permitting non-serializable interleaving of database operations among epsilon transactions. An updated epsilon transaction may export some inconsistency when it updates a data item, and a query epsilon transaction conversely may import some inconsistency. The correctness notion of epsilon serializability is based on bounding the amount of imported and exported inconsistency for each epsilon transaction. Divergence control protocols for centralized [82] and distributed [105, 84] databases ensures bounded divergence. While epsilon-serializability tries to bound the inconsistency on a per-transaction basis, our approach is to bound the inconsistency on a per-site basis. Epsilon transactions require the programmer to specify the inconsistency tolerance of the transactions which can be non-trivial as the data may be modified and read by multiple transactions; each exporting and importing some inconsistency.

Not all database operations necessarily require the same level of consistency. For example, if all transactions in a system commute with one another, then such transactions can be executed in an eventually consistent manner. Transaction classification schemes can be used to identify commutativity between transactions. This approach has been adopted in the context of replicated databases by Kumar and Stonebraker [61], and more recently in the context of geo-replication by Li et al. [67]. Our current approach allows some non-commutative opera-

tions to be executed locally, unless doing so can cause a global inconsistency. We identify conditions under which such local executions are safe by analyzing the transactions.

Yu and Vahdat proposed a continuous consistency model for replicated services [107, 108]. The paper explores the semantic space between traditional strong consistency and eventual consistency. The paper argues for a relaxing consistency as required by applications by bounding the maximum rate of inconsistent access. They used numerical error, order error and staleness as three metrics to capture the consistency spectrum. They rely on anti-entropy algorithms [106, 78, 79] to eliminate or reduce these types of errors. Application developers can bound the maximum distance between the local data image and some final consistent image. We believe it is non-trivial for the application developers to establish such bounds in this distance space, and instead use reasoning to figure out such bounds automatically from the applications transaction code.

Kraska et al. [58] proposed a pay only when it matters approach to consistency. Higher consistency requires a greater cost per transaction, while eventual consistency may have a higher operational cost due to compensation transactions. They partitioned data into three categories: A category items which require strong consistency, C category items which require eventual consistency i.e. temporary inconsistency is acceptable, and B category items whose consistency requirements vary over time. The paper argues that many applications have data items which fall into more than one category, and using a single consistency model for all data items may be overkill. The paper then discusses different statistical policies to adaptively switch the consistency guarantees for

data in category B. While a general policy based on conflict probability has been proposed, specific policies like fixed threshold and demarcation protocols for numeric types [78, 79], and time policies to bound the staleness of data have also been discussed. While the motivation of the paper for consistency ration based on applications consistency requirements is well-aligned with our motivation, our approach is to automatically identify the consistency requirements for data instead of requiring the developer to explicitly specify so. Include the cost of being inconsistent into the optimization problem for figuring out the slack is future work.

Bailis et al. [12] model the expected staleness of data returned by eventually consistent quorum-replicated data stores, and offer SLA-style consistency as an alternative to all-or-nothing consistency guarantees offered by most datastores. They use t-visibility, k-staleness and the combination of th two $< k, t >$-staleness as measures of staleness. t-visibility is the probability of reading a write t seconds after it returns and k-staleness gives the probability of reading one of the last k versions of a data item. Experimentally, they observed that most eventually consistent partial quorum replication schemes frequently deliver consistent data while offering significant latency benefits. Instead of explicitly stating SLA's for consistency, our work tries to infer such SLA's automatically, and enforces them transparently. Flexibility in such SLA's allows the underlying datastore to provide low-latency of eventual consistency without sacrificing consistency from the applications point of view.

Krishnakumar and Bernstein extended the escrow algorithms to propose a generalized site escrow algorithm for replicated databases [60]. Their focus was specific to allocation transactions, and the algorithm used the semantics of the

application and mechanisms of quorum locking and gossip messages to provide a loose synchronization between sites.

TRAPP(Tradeoff in Replication Precision and Performance) [78, 79] was proposed as a new class of replication systems to bridge the gap between systems which return stale query results by executing the query on a cache, and systems which return accurate results by refreshing the cache albeit with poor performance. The paper discusses a specific implementation of TRAPP for aggregation queries in databases. Techniques proposed here have been used by some geo-replication datastores developed by researchers such as Conit-based consistency [108] and consistency rationing [58] to manage staleness efficiently. Yu and Vahdat [106] also proposed techniques for efficiently bounding numerical error in replicated services.

IceCube [56] uses a log-based approach for replica reconciliation. With IceCube, the application is either in an isolated execution phase or in a reconciliation phase. The isolated execution phase executes transactions against a local replica, and produces a tentative final state recorded in a log. The reconciliation phase considers all possible interleaving of the individual site logs including semantics preserving reordering of actions in the log to produce the next initial state for execution phase.

4.6 Geo-Replicated Systems

Finally, we present an overview of other geo-replicated systems which support transaction processing over wide areas and compare them to our approach.

Ideally, replication should be transparent to the users. This goal for transparency is stated as One-Copy Serializabilty(1-SR) [16] criterion. A system is One-Copy Serializable if it behaves like a serial processor of transactions on a one-copy database. Linearizability [52] is an equivalent definition in terms of serial execution of operations as opposed to transactions.

Eric Brewer, in his PODC 2000 keynote [18], proposed Consistency(C), Availability(A) and Partition Tolerance(P) as three desirable systemic properties expected of a distributed system. and observed that all three cannot be satisfied together. This conjecture was later proved by Gilbert and Lynch [43]. The conjecture was followed by a decade of research in developing data stores which relaxed consistency for availability and low-latency.

4.6.1 Weakly Consistent Systems

The term eventual consistency [100] was coined, even before the CAP conjecture, by Terry et al. for the Bayou system. Eventual consistency is a catch all phrase which allows replicas in a system to diverge as long as they may be eventually reconciled. Amazons Dynamo [31] uses a quorum based protocol to implement eventual consistency. Dynamo relies on anti-entropy protocols to eliminate divergence, and ensure in expectation, all reads to be consistent. Dynamo was one of the first eventually consistent systems to be used in production to ensure high-availability with demanding applications. PNUTS [27] provides sequential consistency on a per-key basis. Each object in PNUTS is owned by a primary data-center and all updates to the object must be directed to the primary. Cassandra [64] developed by Facebook provides tunable con-

sistency. Writes in Cassandra are routed to all replicas and the operation blocks until a quorum succeeds. For reads, based on the consistency requested by the client, the system either routes it to the closest replica, or waits for a quorum of responses. Eventually consistent systems required developers to write conflict resolution logic at the application level to deal with inconsistencies arising out of write-write conflicts. This can be non-trivial and burdensome to developers. Walter [98] proposed Parallel Snapshot Isolation as a new property which provided strong guarantees within each site, but weaker consistency across sites. They used preferred sites and counting sets as two techniques to implement Parallel Snapshot Isolation and prevent write-write conflicts. Counting sets are a specific instance of a larger class commutative data types which can be updated in an eventual consistent manner without requiring custom conflict resolution. The notion of causal+ consistency — causal consistency with convergent handling — was first proposed and implemented in the COPS [70, 71] system. While the causal component of the causal+ consistency ensures that causal dependencies between operations is maintained by the data store, the convergent handling component ensures that replicas never permanently diverge. Causal+ consistency is claimed to be the strongest achievable model that is available in the presence of network partitions. Moreover, causal consistency provides useful semantics, disallowing many scenarios that contradict natural human expectations of system behavior. Bailis et al. subsequently pointed to potential dangers of causal consistency by identifying practical issues with implementing causal consistency [10]. First, causal dependencies are not cleanly partitionable and must be sent to all datacenters, limiting sustainable throughput to that of the slowest datacenter. Second, the constraint that each datacenter must buffer writes until it has observed all of the writes dependencies may result in un-

acceptably high visibility latency. They suggest explicit causality as opposed to potential causality to limit the number of causal dependencies to be tracked and mitigate some of the problems above. While the original COPS implementation did not support multi-key transactions, their subsequent implementation Eiger [71] included transactional support and enabled scalable causal consistency for the column-family data model. Eiger still does not support read-write transactions.

4.6.2 Strongly Consistent Systems

Google's Megastore [13] and IBM's Spinnaker [86] pioneered the use of Paxos [65] to build strongly consistent replicated systems. Second-generation data stores like Spanner [28], HyperDex [32] and Calvin [101] have not abandoned consistency, but have rather adopted other techniques to make strong consistency fast. Google's Spanner is a globally distributed and synchronously replicated database, and supports externally-consistent distributed transactions. Spanner is a successor to Megastore, which had relatively poor write throughput. Critical to the implementation of Spanner is the TrueTime API which directly exposes clock uncertainty. This API allows Spanner to assign transactions globally meaningful commit timestamps which reflect a serialization order. Spanner is the first system to provide linearizability at global scale. Spanner uses two-phase locking with wound-wait [92] for R/W transactions, and lock-free read only transactions. They co-exist, as the read-only transactions can be executed at a particular *safe* timestamp in the past.

HyperDex is another strongly consistent datastore, and uses hyperspace

hashing for efficiently searching the datastore, and value-dependent chaining to achieve strong consistency. Value-dependent chaining basically designates a particular server as a leader based on the hash of the objects value, and ensures all updates are propagated through this point leader. This serialization of all updates through the point leader ensures strong consistency.

Calvin [101] uses synchronization to reach consensus on handling transactions beyond transactional boundaries. Such an agreement, once reached, is binding on all nodes. Calvin provides a transaction scheduling and data replication layer which transforms a non-transactional storage system into a linearly scalable shared-nothing database that provides high-availability, strong consistency and full ACID transactions.

MDCC [59] adopts an escrow type approach to get nodes to agree on update options. Similar to Calvin's agreements, a storage node which agrees to an option cannot go back. As opposed to MDCCs escrow transactional approach, we to adopt a escrow site approach. The number of synchronizations required for escrow transactional approach while less than those of two-phase commit, can be further reduced in escrow site approach. Although, the site specific approach requires more semantic understanding of the transactions, and may be limited to only certain data types.

Warranties [69] are guarantees that some assertion holds over the state of a distributed system for a limited period of time. While Homeostasis shares the overall goal of minimizing coordination by enforcing a set of assertions over the distributed state, we differ in how such assertions are extracted and how they are enforced. Homeostasis uses a factorization approach to enforce assertions over a distributed system with minimal communication. However, such

independent enforcement of assertions is only feasible for a limited class of assertions — namely, boolean formulae of linear inequalities.

While Spanner is currently being used by many applications which truly replicate data across geographically distant regions, HyperDex and Calvin's efficacy in the face of high latency links between servers remains to be proved.

CHAPTER 5

CONCLUSION

In this dissertation, we explored opportunities for optimizing OLTP systems by exploiting both transaction semantics as well as transaction workload characteristics. The key idea explored in this dissertation is to exploit flexibility in application requirements to defer the materialization of certain changes made by transactions until such materialization is necessary to preserve correctness. In the context of resource allocation applications, such deferred execution of transactions creates a window for optimizing allocation of resources. On the other hand, in the context of wide area transaction processing, such deferred propagation of transactional writes allows us to minimize the amount of cross-site communication as well as removes communication latency from critical path of transaction execution. We summarize the key ideas used in developing such lazy transaction execution systems below.

Semantic Analysis: The first key idea explored in this dissertation is to use transaction semantics to allow a greater degree of concurrency between transactions. Broadly we adopt two different approaches for doing such semantic analysis of transactions. The first approach is to restrict the expressivity of the transactions to simplify semantic analysis as in the case of resource transactions. The other approach is to use program analysis techniques to infer the sensitivity of transactions to input databases and use such sensitivity analysis as coarse grained transaction semantics sufficient for deciding safety of concurrent execution.

Deferred Writes: The second key idea explored in this dissertation is to defer writes performed by the transactions until applying such writes becomes

necessary to ensure semantically correct execution of other transactions. In the case of resource allocation applications, such deferred execution is achieved by implicitly maintaining multiple possible worlds using quantum databases. Deferred assignment of resources allows more optimal resource allocation in high contention environments. On the other hand, in the case of wide area transaction processing, deferred propagation of transactional writes not only reduces the execution latency by removing communication between sites from the critical path of execution, but also permits batching multiple such writes together thereby reducing the number of rounds of communications.

Workload Adaptation: The third key idea is to exploit offline and online knowledge about transactional workload to improve performance of OLTP systems. Specifically, through our system, Homeostasis, we have established how flexibility in the application semantics in conjunction with knowledge of transactional workload can be exploited to optimize communication between sites in a distributed system. To the best of our knowledge, this dissertation is the first work which takes a principled approach towards deeper integration of workload characteristics into concurrency control protocols.

BIBLIOGRAPHY

[1] http://dev.mysql.com/.

[2] http://www.cs.cornell.edu/bigreddata/youtopia/.

[3] http://tpc.org/tpcc.

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

[5] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In *ICDE*, pages 67–78, 2000.

[6] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR*, pages 249–260, 2011.

[7] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. Fast and simple relational processing of uncertain data. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 983–992, Washington, DC, USA, 2008. IEEE Computer Society.

[8] Lyublena Antova, Christoph Koch, and Dan Olteanu. $10^{(10^6)}$ worlds and beyond: efficient representation and processing of incomplete information. *The VLDB Journal*, 18(5):1021–1040, October 2009.

[9] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination-avoiding database systems. http://arxiv.org/pdf/1402.2237.pdf.

[10] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the 2012 ACM Symposium on Cloud Computing (SOCC 2012)*, 2012.

[11] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32, March 2013.

[12] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.*, 5(8):776–787, April 2012.

[13] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[14] Daniel Barbará-Millá and Hector Garcia-Molina. The demarcation protocol: a technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, July 1994.

[15] 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.

[16] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[17] Nikolaj Bjørner. Engineering theories with z3. In *Proceedings of the First international conference on Certified Programs and Proofs*, CPP'11, pages 1–2, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[19] O. S.F. Carvalho and G. Roucairol. On the distribution of an assertion. In *PODC '82*, pages 121–131, 1982.

[20] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[21] Jianjun Chen, Ashwin Machanavajjhala, and George Varghese. Scalable social coordination with group constraints using enmeshed queries. In *Conference on Innovative Data Systems Research*, 2013.

[22] Wei Chen, Alan Fekete, Paul Greenfield, and Julian Jang. Implementing isolation for service-based applications. In *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part I*, OTM '09, pages 365–372, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *PLDI*, 2008.

[24] Alvin Cheung, Sam Madden, Armando Solar-Lezama, Owen Arden, and Andrew C. Myers. Statusquo: Making familiar abstractions perform using program analysis. In *Sixth Biannual Conference on Innovative Database Systems Research (CIDR13), Asilomar*, 2013.

[25] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 931–942, New York, NY, USA, 2014. ACM.

[26] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Inferring sql queries using program synthesis. *CoRR*, abs/1208.2013, 2012.

[27] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[28] James C. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, pages 251–264, 2012.

[29] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. TACAS'08/ETAPS'08, 2008.

[30] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[31] Giuseppe DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[32] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: a distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[33] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Vardi. *Reasoning about Knowledge*. MIT Press, 2004.

[34] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[35] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 15–26, New York, NY, USA, 2014. ACM.

[36] Shel Finkelstein, Thomas Heinzel, Rainer Brendle, Ike Nassi, and Heinz Roggenkemper. Transactional intent. In *CIDR*, 2011.

[37] Zhaohui Fu and Sharad Malik. On solving the partial max-sat problem. SAT'06, 2006.

[38] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, June 1983.

[39] Hector Garcia-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. Coordinating multi-transaction activities. Technical report, College Park, MD, USA, 1990.

[40] Hector Garcia-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. Modeling long-running activities as nested sagas. *Data Eng.*, 14(1):14–18, 1991.

[41] Hector Garcia-Molina and Kenneth Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, 1987.

[42] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS fast path. *Database Engineering*, 8(2):63–70, June 1985.

[43] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[44] Todd J. Green and Val Tannen. Models for incomplete and probabilistic information. In *Proceedings of the 2006 international conference on Current Trends in Database Technology*, EDBT'06, pages 278–296, Berlin, Heidelberg, 2006. Springer-Verlag.

[45] Paul Greenfield and Dean Kuo. Isolation support for service-based applications: A position paper. In *Third Biannual Conference on Innovative Database Systems Research (CIDR07), Asilomar*.

[46] Nitin Gupta, Lucja Kot, Sudip Roy, Gabriel Bender, Johannes Gehrke, and Christoph Koch. Entangled queries: enabling declarative data-driven coordination. In *SIGMOD*, 2011.

[47] Nitin Gupta, Milos Nikolic, Sudip Roy, Gabriel Bender, Lucja Kot, Johannes Gehrke, and Christoph Koch. Entangled transactions. In *VLDB*, 2011.

[48] Ashish Gupta et al. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *VLDB*, 2014.

[49] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *PLDI*, 2012.

[50] Pat Helland and Don Haderle. Engagements: Building eventually ACiD business transactions. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2013.

[51] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPOPP*, pages 207–216, 2008.

[52] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[53] Krytof Hoder, Nikolaj Bjrner, and Leonardo Moura. μz an efficient engine for fixed points with constraints. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 457–462. Springer Berlin Heidelberg, 2011.

[54] Tomasz Imieliński and Witold Lipski, Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, September 1984.

[55] Julian Jang, K. Fekete, and Paul Greenfield. Delivering promises for web services applications. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 599–606, 2007.

[56] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. In *PODC*, Newport, RI, USA, 2001.

[57] Lucja Kot, Nitin Gupta, Sudip Roy, Johannes Gehrke, and Christoph Koch. Beyond isolation: Research opportunities in declarative data-driven coordination. *SIGMOD Record*, 39(1):27–32, 2010.

[58] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, August 2009.

[59] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: multi-data center consistency. In *EuroSys*, 2013.

[60] Narayanan Krishnakumar and Arthur J. Bernstein. Abstract high throughput escrow algorithms for replicated databases* (extended abstract).

[61] Akhil Kumar and Michael Stonebraker. Semantics based transaction management techniques for replicated data. In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, SIGMOD '88, pages 117–125, New York, NY, USA, 1988. ACM.

[62] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[63] Gabriel M. Kuper, Leonid Libkin, and Jan Paredaens, editors. *Constraint Databases*. Springer, 2000.

[64] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[65] Leslie Lamport. Paxos Made Simple. *SIGACT News*, 32(4):51–58, December 2001.

[66] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, July 2006.

[67] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguica,

and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, pages 265–278, 2012.

[68] Changbin Liu, Lu Ren, Boon Thau Loo, Yun Mao, and Prithwish Basu. Cologne: A declarative distributed constraint optimization platform. *PVLDB*, 5(8):752–763, 2012.

[69] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 503–517, Seattle, WA, April 2014. USENIX Association.

[70] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[71] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, 2013.

[72] Nancy A. Lynch. Concurrency control for resilient nested transactions. In *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '83, pages 166–181, New York, NY, USA, 1983. ACM.

[73] Keith Marzullo and Gil Neiger. Detection of global state predicates. In *In Proc. 5th Int'l. Workshop on Distributed Algorithms (WDAG '91*, pages 254–272. Springer-Verlag, 1991.

[74] Alexandra Meliou and Dan Suciu. Tiresias: the database oracle for how-to queries. In *SIGMOD*, pages 337–348, 2012.

[75] Gerome Miklau and Dan Suciu. A formal analysis of information disclosure in data exchange. *J. Comput. Syst. Sci.*, 73(3):507–534, 2007.

[76] E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, Cambridge, MA, USA, 1981.

[77] Marian H. Nodine and Stanley B. Zdonik. Cooperative transaction hier-

archies: transaction support for design applications. *The VLDB Journal*, 1:41–80, July 1992.

[78] Chris Olston, Boon Thau Loo, and Jennifer Widom. Adaptive precision setting for cached approximate values. In *SIGMOD*, pages 355–366, 2001.

[79] Chris Olston and Jennifer Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *VLDB*, pages 144–155, 2000.

[80] Patrick E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, December 1986.

[81] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, December 1976.

[82] C. Pu, W. Hseush, G.E. Kaiser, Kun-Lung Wu, and P.S. Yu. Distributed divergence control for epsilon serializability. In *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, pages 449–456, 1993.

[83] Calton Pu, Gail E. Kaiser, and Norman C. Hutchinson. Split-transactions for open-ended activities. In *Proceedings of the 14th International Conference on Very Large Data Bases*, VLDB '88, pages 26–37, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[84] Calton Pu and Avraham Leff. Replica control in distributed systems: as asynchronous approach. In *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, SIGMOD '91, pages 377–386, New York, NY, USA, 1991. ACM.

[85] Krithi Ramamritham and Calton Pu. A formal characterization of epsilon serializability. *IEEE Trans. on Knowl. and Data Eng.*, 7(6):997–1007, December 1995.

[86] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4(4):243–254, January 2011.

[87] Andreas Reuter. Concurrency on high-traffic data elements. In *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '82, pages 83–92, New York, NY, USA, 1982. ACM.

[88] Andreas Reuter and Helmut Wächter. The contract model. *IEEE Data Eng. Bull.*, 14(1):39–43, 1991.

[89] Peter Revesz. *Introduction to constraint databases*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[90] J. Roberts, T. Collier, P. Malu, and K. Srinivasan. The tentative hold protocol. W3C Note, www.w3.org/TR/tenthold-2/., Nov 2001.

[91] J. Roberts and K. Srinivasan. The tentative hold protocol. W3C Note, www.w3.org/TR/tenthold-1/., Nov 2001.

[92] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, June 1978.

[93] J. B. Rothnie, Jr., P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. Reeve, D. W. Shipman, and E. Wong. Introduction to a system for distributed databases (sdd-1). *ACM TODS*, 1980.

[94] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for horn-clause verification. In *CAV*, pages 347–363, 2013.

[95] Bart Selman. Stochastic search and phase transitions: AI meets physics. In *IJCAI (1)*, 1995.

[96] Marc Shapiro, Nuno Preguia, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical report, INRIA, Rocquencourt, France, 2011.

[97] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *PLDI*, volume 43, pages 136–148, 2008.

[98] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[99] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[100] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[101] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[102] Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. *ACM SIGPLAN Notices*, 43(6):125–135, 2008.

[103] Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, volume 45, pages 327–338, 2010.

[104] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems*. Morgan Kaufmann, 2002.

[105] Kun-Lung Wu, Philip S. Yu, and Calton Pu. Divergence control for epsilon-serializability. In *Proceedings of the Eighth International Conference on Data Engineering*, pages 506–515, Washington, DC, USA, 1992. IEEE Computer Society.

[106] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *OSDI*, pages 305–318, 2000.

[107] Haifeng Yu and Amin Vahdat. Efficient numerical error bounding for replicated network services. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 123–133, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[108] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, August 2002.

[109] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, pages 276–291, 2013.