SCALABILITY IN COORDINATED TRANSACTION MANAGEMENT

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

Nitin Gupta

January 2012

SCALABILITY IN COORDINATED TRANSACTION MANAGEMENT

Nitin Gupta, Ph.D.

Cornell University 2012

Transaction processing systems are commonly employed in a wide range of applications, such as travel booking, financial trading, and online games. Typically, these applications are used by thousands of users simultaneously, and many a times, users attempt to coordinate on data values. In this dissertation, we argue that the current support for coordination in such systems is ad-hoc and not scalable. We propose simple, efficient, and scalable abstractions for coordination in transaction processing systems.

This dissertation comprises of three papers – Scalability in Virtual Environments, Declarative Data-driven Coordination, and Entangled Transactions. In each of these papers, we look at scalability of coordinated transaction management from a unique perspective that can remarkably alter the manner in which coordination is perceived by both the users and developers of the aforementioned applications.

In the first paper, we address scalability in virtual environments. Virtual environments are software systems in which users interact with each other in real-time within some shared environment. Current virtual environments, however, are unable to support a large number active users. The scalability problems arise in part because of the need to maintain consistency between all the players. In this paper, we propose a protocol that actively replicates actions, and show that replicating actions, as opposed to techniques that replicate data, allows for highly scalable virtual environments. We also propose an optimization of the AB-protocol that guarantees minimal execution at client machines.

In the second paper, we explore declarative data-driven coordination. We propose Entangled Queries, a novel abstraction for coordination in databases. Entangled Queries provides the user with a simple but powerful declarative method to coordinate with other users. In addition to introducing Entangled Queries, we propose an efficiently enforceable syntactic safety condition that we argue is at the sweet spot of expressiveness and application requirements.

In the last paper of this dissertation, we introduce *entangled transactions*. Entangled Transactions are units of work performed within a database management system against a database. Although such transactions look similar to classical transactions, they do not run in isolation and communicate with each other via entangled queries. In this paper, we look at an abstract model for Entangled Transactions and investigate interesting system issues that arise in their implementation.

BIOGRAPHICAL SKETCH

Nitin Gupta started coding at a young age of seven, and followed his passion to college. He joined the Computer Science department at the Indian Institute of Technology Bombay to learn system design and algorithms, but discovered a whole new world of technology and science that strengthened his interest in computers. After having worked on search algorithms with Prof. S. Sudarshan, Nitin did an internship at University of British Columbia in Summer 2005 followed by another research internship at INRIA Futurs in Summer 2006, where he worked on algorithms for search in unstructured databases.

Nitin joined the Computer Science department at Cornell in Fall 2006. He began working with Prof. Johannes Gehrke on languages for software personalization, and spent the next few years working on scalability in games. He also worked with Prof. Christoph Koch on a new abstraction for coordination in databases that was awarded the best paper at SIGMOD 2011. Nitin defended the work presented here as his Ph.D. dissertation in September 2011, and is now heading to work in the Silicon Valley.

To my parents.

ACKNOWLEDGMENTS

I would first and foremost like to thank my co-authors of the work presented in this dissertation for the deeply intellectual conversations that have contributed immensely to my research at Cornell. These people, in alphabetical order, are Tuan Cao, Bailu Ding, Michaela Goetz, Lucja Kot, Sudip Roy, Marcos Vaz Salles, Ben Sowell, Guozhang Wang, Walker White, Fan Yang, and Tao Zou.

I have benefited most at Cornell from the teachings of my adviser, Johannes Gehrke. I remember that when I started working with him, he categorically stated that I not only do good research, but become a good researcher. In the last few years, he has guided me to accomplish that goal, and has helped me develop a progressive attitude in life that I will cherish forever.

In addition, there have been two faculty members who I have worked closely with – Alan Demers and Christoph Koch. There have been times when I was clueless of the direction of my research; but Al and Christoph stepped forward and helped me take the next step. They have motivated me to address and solve hard problems and much of my growth at Cornell is attributed to their kind guidance.

Furthermore, I would to thank my committee members, Andrew Myers and Robert Bloomfield. Andrew has been very insightful and has helped steer my research in the correct direction. He has been instrumental in helping me focus on interesting problems. I have worked with Robert over the last one year on different projects and he has helped me develop a good sense of economics and business. His knowledge on virtual worlds has been a significant contribution in streamlining my research.

Life outside 4104 Upson would not have been the same without my friends. Their enthusiasm and support made my life more exciting and enjoyable. I want to thank Anirikh Chakrabarti and Hamish Chandola for all the fun and the many experimental cooking experiences.

My family has been a source of inspiration for me. They have seen and born the ups and downs of my life, and I have fallen back on them whenever I was in need. Their encouragement has kept me going. I thank you mom, dad, *bhaiya*, and *bhabhi* for always being there.

Finally, but most importantly, I want to thank Pooja. It is very hard for me to imagine the situation of a girl who traveled across the world to be as close to her husband as she can, adjust in a new country, cope up with work pressure, and yet never complain. Thank you Pooja for being there all this while.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Scalability [ˌskeɪləˈbɪlɪtɪ]

- n the ability of something, esp a computer system, to adapt to increased demands

Coordination [koʊˌɔrdnˈeɪʃən]

- n harmonious combination or interaction, as of functions or parts

1.1 Overview

A transaction processing system is a type of information system that is used to collect, store, modify, and retrieve the transactions of an application. The definition follows from that of a transaction which is an event that generates or modifies data that is eventually stored in an information system. Transaction processing systems are commonly employed in a wide range of applications, such as travel booking, financial trading, and online games. Typically, these applications are used by hundreds and thousands of users simultaneously, and many a times, users attempt to coordinate on data values. In this dissertation, we argue that the current support for coordination in such systems is ad-hoc and not scalable. We propose simple, efficient, and scalable abstractions for coordination in transaction processing systems.

This dissertation comprises of three papers – Scalability in Virtual Environments, Declarative Data-driven Coordination, and Entangled Transactions. In each of these papers, we look at scalability in coordinated transaction management from a unique perspective, one that can remarkably alter the manner in which coordination is perceived by both the users and developers of the aforementioned applications.

1.2 Scalability in Virtual Environments

In the first paper, we address scalability in virtual environments (VE). Virtual environments are software systems in which users interact with each other in real-time within some shared environment. With increasing popularity, VEs are poised to be the next wave of digital entertainment, with Massively Multiplayer Online Games (MMOs) a very popular instance. Current MMO architectures are server-centric in that all game logic is executed at the servers of the company hosting the game. The server-centric architecture, however, does not scale both in the number of users and their interactions with the environment, primarily because MMOs require realistic graphics and game physics – computationally expensive tasks that are currently computed centrally.

In this paper, we argue that the client-side architecture is an optimal candidate for VE architectures, and propose a distributed *action based* protocol for virtual environments. The key feature of our protocol is active replication of actions. We show that replicating actions, as opposed to the currently popular techniques that replicate data, allows for highly scalable virtual environments. We also propose an optimization of the AB-protocol that guarantees minimal execution at client machines. Our protocols allow us to explore the tradeoff between scalability, computational complexity, and consistency. We investigate our proposal both theoretically and through a comprehensive experimental evaluation.

1.3 Declarative Data-driven Coordination

In the second paper, we propose an abstraction for coordination in transaction processing systems. This paper introduces *entangled queries*, a declarative language that extends SQL by constraints that allow for the coordinated choice of result tuples across queries originating from different users or applications.

It is nontrivial to define a declarative coordination formalism without arriving at the general (NP-complete) Constraint Satisfaction Problem from AI. In this paper, we propose an efficiently enforcible syntactic *safety* condition that we argue is at the sweet spot where interesting declarative power meets applicability in large scale data management systems and applications.

The key computational problem of declarative data-driven coordination is to match entangled queries to achieve coordination. We present an efficient matching algorithm which statically analyzes query workloads and merges coordinating entangled queries into compound SQL queries. These can be sent to a standard database system and return only coordinated results. We present the overall architecture of an implemented system that contains our evaluation algorithm; we also evaluate the performance of the matching algorithm experimentally on realistic coordination workloads.

1.4 Entangled Transactions

In the last paper of this dissertation, we propose *entangled transactions*. Entangled Transactions are units of work performed within a database management system against a database. Although such transactions look similar to classical transactions, they do not run in isolation and communicate with each other via entangled queries.

We motivate entangled queries by looking at an example. Assume that two friends, Mickey and Minnie, wish to travel to Los Angeles on the same flight and stay at the same hotel. Their arrival date is flexible, but their departure date is fixed. They start by jointly selecting a suitable flight. Once they know the flight number, and consequently their date of arrival in Los Angeles, they will try to make joint hotel reservations. With existing mechanisms, they can use entangled queries to coordinate on the choice of the flight and then on their choice of hotel. These queries, however, need to be embedded within a larger code unit that Mickey and Minnie separately execute and populate with their constraints such as the class of the hotel or airline restrictions. Once both their individual *entangled transactions* have been submitted, the system needs to match them up, execute the associated logic, and guarantee "transaction-like" semantics for this execution.

In this paper, we first introduce a novel semantic model for entangled transactions that comes with analogues of the classical ACID properties. We show that despite the interaction among them, each of the entangled transactions represents a logical unit of work on its own, *and* that this work is dependent on input from other transactions in the system. We also show how our model for entangled transactions extends to transactions that contain more than one entangled query.

We then discuss execution models for entangled transactions and select a concrete design motivated by application scenarios. With a prototype system that implements this design, we show experimental results that demonstrate the viability of entangled transactions in real-world application settings.

CHAPTER 2

SCALABILITY IN VIRTUAL ENVIRONMENTS

2.1 Introduction

Networked virtual environments (VE) are software systems in which users interact with each other in real-time within some shared virtual environment. [21]. Virtual worlds are typically designed to create a very high degree of immersion. Many feature 3D graphics and stereo sound, and have extremely interactive environments. But the primary selling point of many virtual worlds is the large number of players that they can support. In MMOs like *World of Warcraft* it is common for groups of up to 40 players to work cooperatively in a "raid" [73]. Other online virtual worlds like *Habbo Hotel* [63] market themselves as social-networking environments, and must support large parties or other social events online. While high-bandwidth, low-latency internet is now becoming ubiquitous, this is not enough to solve the scalability issues that VEs are beginning to encounter.

These scalability problems arise in part because of the need to maintain consistency between all the players. In the best case, inconsistency may just lead to transient visible artifacts with no long-term consequences. However, in practice, it can easily cause much more serious problems, like objects being lost or duplicated during a financial transaction. In addition to degrading the realism of the virtual world, consistency violations are a major source of security problems in VEs [31]. To maintain consistency, all VEs have a transaction management layer that employs a commercial database. Every interaction in the VE encapsulates a transaction that is executed on the database. In essence, as players interact with the virtual environment, they send transactions to the database at an extremely high rate. The transaction management layer, therefore, is affected by severe scalability problems. Even the fastest MMOs cannot handle more than about 10 frames per second [14] through their database transaction layer.

Figure 2.1 illustrates the scalability-complexity tradeoff for a sample of current VEs. Social games such as Farmville are highly scalable because user interactions involve only simple updates that rarely conflict. MMO Games with a static environment such as *World of Warcraft* require comparatively more computational resources, leading to a drop in scalability [43]. Simulators, particularly military simulators such as SIMNET, are even more “real” than virtual worlds, in that players can interact with the virtual environment (e.g., destroy buildings); the result is even less scalability [40]. Finally, user-designed virtual worlds such as *Second Life* [42] allow objects to be created, modeled, and scripted by the users at run-time. This flexibility comes with high computational cost; for example, the resulting scalability of *Second Life* is on the order of at most 25-30 players per server [37]. If the player-to-server ratio of collaborative software were possible in a VE with the flexibility and degree of immersion of virtual worlds, this would allow for a user experience beyond the reach of current systems.

The desire to support more players in complex environments has spawned research both in distributed system architectures [8, 17, 24, 61] and database management systemsdatabase management systems [5, 29].

A problem with the methods adopted in distributed systems is that the user interactions, such as shooting a person, are tied to character visibility. However, in real VEs, players often interact in complex and subtle ways beyond visibility. For example, suppose we have a fantasy MMO designed to support a large number of players. A classic feature for such a game is a “scrying spell” that

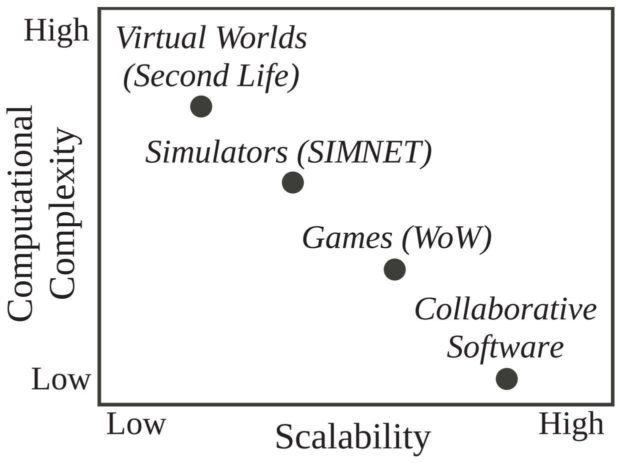


Figure 2.1: Scalability versus Complexity

allows a healer to identify and heal the most wounded ally in a crowd [3]. During combat, the result of this spell affects many users beyond the visibility of the healer, as the health of each player is continually changing. The range and nature of such a spell makes character-visibility partitioning useless.

Database management systems, on the other hand, are not ideal candidates for VE engines for the following reasons. First, database management systems require that significant parts of the application logic be executed on the server side. As a result, the scalability of an application is strongly related to the computational footprint of a single machine. One could argue that server-side computing platforms such as commercial cluster instances are more powerful; but given that the interactions in VEs are computationally expensive, server-side computing platforms are unable to handle the load for millions of players. Second, distributed databases work by either by partitioning or replicating and synchronizing the database. While database partitioning is outright unacceptable for true world VEs, inter-node communication in data replication has additional latency that negatively affects gaming experience.

Fortunately, virtual worlds have a lot of semantic information that can be leveraged for scalable consistency. Virtual worlds and simulations are essentially high-dimensional databases where the attributes can change only in predictable ways [70]. For example, in a fantasy MMO game, health is itself an attribute that changes as a player is damaged. By examining semantic information such as the maximum damage that an attack transaction can cause, we can predict the ways in which the health attribute can change, and exploit this semantic information to reduce the number of messages needed to maintain a consistent state among the many distributed clients.

In this paper, we propose a distributed model for virtual environments that achieves massive scalability. Our model inherits concepts from distributed databases, where the game play and transaction processing take place at the client machine, thereby reducing the computation performed at the server. The key feature of our model is its novel transaction model that eliminates unwarranted inter-node communication in replicated TPS to reduce latency, and exploits application semantics to reduce the number of messages needed to maintain consistency. Our model assumes realistic restrictions on the interaction between participants located in different parts of the world. We also show how our model can be scaled to a massive number of participants.

Outline of the Paper

We begin the paper, in Section 2.2, by arguing that client-side architecture is an optimal candidate for VEs. Further, our paper continues with the following contributions.

- In Section 2.3, we introduce the AB-protocol that works by actively replicating actions in the virtual environment among client replicas.

- In Section 2.4, we give the MinAB-protocol that has minimizes computation in the AB-protocol thus making is massively scalable.
- In Section 2.5, we explore boundedness of the MinAB-protocol. We also detail certain application semantics to provide theoretical bounds that show the scalability of our approach, and present several techniques that leverage spatial properties of VEs to optimize our protocols.
- In Section 2.6, we present an experimental evaluation using both simulation and real experiments demonstrating the effectiveness of our new protocol.

We conclude with a discussion in Section 2.7.

2.2 Background

As discussed in the previous section, consistency is important to VEs. Realistic gameplay requires that everyone share a single view of the virtual environment, the *world state*. Virtual environments typically represent the world state in a database [4]. Any interaction in the environment can be perceived as a database transaction: making an observation is a database query about the state of the world, and a change in state is a database update. However, because of limited throughput of commercial databases, most VEs use them only to commit and read at periodic checkpoints [10]. For real-time interactions, they generally implement their own in-memory transaction layer on top of the database [71]. This design decision is not because database transactions are unsuited to the task; rather, it is because existing commercial databases are not optimized for the type of in-memory processing that VEs need for real-time performance [14].

In this section, we look at popular database architectures used by virtual environments. We argue that a client-side architecture with a central audit server is a good design choice.

2.2.1 Server-side Architectures

In practice, databases are often architected such that most of the processing happens at the server. The server may be a cluster of machines, in which case the computation is distributed among these machines. The clients in such server-side architectures are analogous to I/O devices for the purpose of the game play [16].

Server-side architectures are instances of master-slave replication. The server (also called the "master") executes all transactions, and the write sets are then propagated to all other clients ("slaves"), which update data in the same order so as to guarantee convergence of their final states with that of the master. Log shipping [32, 53] is a popular technique to send logs to client machines.

In order to scale with an increasing number of users and handle the heavy cost of computation, virtual environments commonly use techniques that permit database architectures with low throughput rates. Three such techniques are:

Zoning. Zoning refers to the technique of geographically partitioning ("tiling") the virtual environment into areas small enough for a single server to handle. Commercial MMOs have only recently adopted the idea of dynamic zoning [14]. While dynamic zones are more flexible than traditional zones, they still restrict player actions to a geographic area.

Sharding. Zoning works well to about a few dozen servers, which translates into a few thousand players for most virtual worlds. In order to scale beyond a few thousand players, MMO companies instantiate completely separate instances of the virtual world called *shards*.

Instancing. Unlike sharding, *instancing* is confined to small partitions of the virtual environment. An instance is essentially a private zone into which no players may enter except those that originally spun off the instance. In *World of Warcraft*, instancing is used heavily for dungeons that are intended to be personal experiences [74].

All of the above techniques split the user base, degrading the “massive” multiplayer experience [33]. For example, sharding and instancing prevent large groups of users from working together by design, while zoning collapses if too many users crowd into a single zone [67]. Users often have difficulty finding their real life friends in such MMOs. Some virtual worlds even require the users to pay if they want to play with someone of their choice [65]. Therefore, MMO companies are still struggling to meet the scalability requirements demanded by their user base.

2.2.2 Client-side Architectures

Alternatives to the server-side architectures are the *client-side architectures* [4], in which computation is distributed among client machines in order to achieve scalability. Distributing computation between clients has the potential not only to reduce load on the central server, but also to leverage capabilities of the client machines.

Peer-to-peer

While P2P architectures seem to be the natural choice for client-side architectures, there are both technical and non-technical reasons to not choose P2P.

First, *strongly consistent* P2P architectures do not scale because they use protocols such as Paxos [38] or Virtual Synchrony [7] to enforce a consistent total order of events across all participants. Examples of virtual worlds that use a peer-to-peer architecture include Reality Build For Two [8] and MR Toolkit [61]. Both these VEs maintain consistent state among N workstations by sending a point-to-point message to each of the workstations for every single state change. This approach yields $O(N^2)$ update messages during every simulation step, and this does not scale.

However, there is an even stronger, non-technical reason to rely on a server-side architecture. Virtual environments are developed and operated by companies that have a vital interest in exerting total control over the virtual world, even if that means investing in server hardware. In many virtual worlds, players pay real money both to participate and for game content; hence the MMO company has an obligation to provide uninterrupted service. Additionally, the absence of audit logs makes cheating a major concern for peer-to-peer MMOs [28]. For these reasons, companies desire to have all content stored securely and persistently by a trusted authority.

Distributed with Central Auditor

An architecture that has client-side computing with a central audit server strikes a balance between preserving the interests of the MMO companies in exerting control, scalability of the system, and alleviating the problems of no centralized control compared to a P2P architecture. Thus, we will adopt it in the remainder of the paper.

A client-side architecture with a central audit server consists of a server to which all clients connect. Without loss of generality, we assume that the clients are replicas and run identical VE software, which we refer to as *client programs*. The client program contains the actual virtual world logic. Clients initiate and process units of code called *actions* in the environment.

Definition 2.2.1. ***Action.*** *An action is a stored procedure call that encapsulates a transaction. It can be perceived as the code that specifies both application logic and database queries. Any action must adhere to the ACID properties that are associated with the encapsulated transaction.*

An example of an action is the procedure call to move a player in the virtual environment that involves first a query of the player's position and surroundings, followed by a check for conflicts on the movement, and finally an update of the state. In this paper, we assume that each action consists of exactly one atomic operation. This is merely for simplicity of exposition, and has no effect on the techniques proposed. Though processing actions in the client program may raise security issues, a lot of prir research already exists for developing non-hackable clients [59, 60]. As an added security measure, the servers can also log MMO statistics to detect any cheating or security threat [28].

The key component of a client-side architecture is its consistency protocol. Since the transactions are executed at the clients, a protocol needs to be established between clients and server that ensures consistency and durability of data. Commonly used protocols generally fall into one of three families, each with its own subtleties, variations, and costs.

Lock Based Protocols. Distributed locking is a popular family of protocols used to provide consistency. To process a transaction, a client must first acquire global locks on the objects read and written by the transaction. This can be implemented by having all clients in the system agree on granting a lock request, or by managing locks at the server-side. By virtue of two message exchanges per transaction, distributed locking has a high overhead for transaction latency. This can significantly detract from achievable transaction throughput required in virtual environments. Furthermore, the consistence resolution in two-phase locking is object based, while many consistency problems in VEs are semantic. The virtual world designer is forced to map every single consistency issue in the world to an object access, which is not always easy to do.

Timestamp Based Protocols. Timestamp optimistic concurrency control is a well-known alternative to locking. Here, we associate a version with every object, and a timestamp with every transaction; the timestamp can be assigned by the server. Clients optimistically execute tentative actions against their local, possibly stale versions of objects. The server integrates the local, transactional histories submitted by clients into a global multi-version history. Since the server makes commit and abort decisions, the server must understand game-specific logic and perform possibly expensive operations in order to resolve conflicts. For example, any change in the read set of a transaction, such as some player moving, would potentially cause the transaction to abort. In order to neglect irrelevant changes, the server must implement a significant part of the application logic that specifies what combination of movements are valid.

Object Ownership Based Protocols. Object ownership differs from lock-based protocols in that each object is owned and managed by exactly one client, known as the object owner. Other clients are allowed to cache a version of the object, but are not allowed to make modifications to its state. RING [17], Cyberwalk [49], and WAVES [30] are three popular systems using such a protocol. Variations of such protocols allow non-owners to obtain "leases" from the master for a particular data item. In order to allow object contention in such a protocol, applications are either degraded to a lower level of consistency, or are forced to employ timestamp-based serializability [6], resulting in unacceptable response time for VEs.

RING [17] and DIVE [24] are VEs that use a distributed architecture with a central auditor. They handle message filtering by sending all updates to the central server. The server tracks the location of each client in the virtual world, and thus determines the updates that a client would be interested in. In Section 2.4.1, we show that this filtering of updates based on who can "see" the client leads to inconsistency. Two recent proposals in the database literature [5, 66] also use a distributed architecture with central auditor. [66] introduces concurrency in deterministic distributed transaction processing systems. Fundamentally, the proposal increases availability and redundancy of the database but does very little towards scalability. Hyder [5] is an optimistic concurrency control framework that uses intention logs of actions in messages as a substitute for stored procedure calls. However, the system is susceptible to high abort rates by virtue of its optimistic framework.

A common characteristic in all popular server-side and client-side architectures is that while the data is either partitioned or replicated, processing is always partitioned, i.e. any given action is always executed by only one machine. What would be the scalability and consistency properties of an architecture in which each action is executed on a few (or all) client machines? In this paper, we seek to explore the effect resulting from replication of computation.

2.3 Action Based Protocols

Recent developments in database research have given way to models that are characterized by fast execution of transactions across all replicas of the client [66, 21]. In order to better understand how transaction processing is replicated at clients, we next describe our action based protocol (AB-protocol) that lays the foundation for this paper.

Action based protocols are a family of protocols designed for client-side architectures that work by replicating both data and actions across client machines. In AB-protocols, the messages passed between the clients and the server primarily consist of *actions*. An action a consists of a read set $RS(a)$, a write set $WS(a)$ and the code that needs to be executed to compute values for $WS(a)$ given values for $RS(a)$. For simplicity of exposition we also assume that $RS(a) \supseteq WS(a)$. This allows us to drop the distinction between read sets and write sets and focus on intersecting read sets in our discussion and protocols. The state of the virtual world is a database of objects, the *world state*. Each client program maintains two versions of the world state: an optimistic version `CO` and a consistent version `CS`. The player always sees `CO` and therefore might witness effects of uncommitted transactions; however, `CO` is reconciled periodically with `CS` to correct the game play.

The basis of AB-Protocol is its execution model that has the property that any action a_n's outcome is uniquely determined by the database's initial state and a totally ordered series of previous actions $a_0, a_1, ..., a_{n-1}$. Algorithms 1 and 2 give the pseudo-code for the server and the client. To execute an action a, a client first executes a on `CO` to get the *optimistic evaluation* v, which we denote $v = a(\texttt{CO})$. It simultaneously submits a to the server for serialization. Upon submission, the server attaches a monotonically increasing sequence number to a and broadcasts it to all clients. In effect, the client receives a serialized stream of actions originating at *all* clients. It executes them, in order, on `CS`. The results of applying locally originated actions to `CO` and `CS` are compared, and disagreements are reconciled if necessary.

Algorithm 1: Client-Side Protocol

```
1 Require: $Q$ a queue of unreconciled optimistic evaluations $\langle a_i, v_i \rangle$ where
  $v_i = a_i(\texttt{CO})$
 1: Create action $a$ and apply to get $v = a(\texttt{CO})$
 2: Add $\langle a, v \rangle$ to $Q$ and send $a$ to server.
 3: Wait to receive action $b$ from the server.
 4: if $b$ is not an action in $Q$ then
 5:    Apply action $b$ to $\texttt{CS}$
 6:    for each write $x \leftarrow v$ performed by $b$ do
 7:       if $x \notin WS(Q)$ then
 8:          Perform the write $x \leftarrow v$ on $\texttt{CO}$
 9: else if $b = a_1$ then {$b$ must be head of $Q$}
10:    Apply $a_1$ to $\texttt{CS}$ to get result $u = a_1(\texttt{CS})$
11:    if $u = v_1$ then {optimistic evaluation okay}
12:       Remove $\langle a_1, v_1 \rangle$ from $Q$
13:    else
14:       Reconcile $\texttt{CO}$ with $\texttt{CS}$ via Algorithm 3
```

Algorithm 2: Server-Side Protocol

```
Require: $Q$ is a global queue of actions
Require: $pos_C$ is index of action last sent to client $C$
 1: Wait for action $a$ from client $C$
 2: Timestamp $a$ and put it into $Q$
 3: $pos(a) \leftarrow$ index of $a$ in $Q$
 4: Send $C$ all actions in $Q$ between $pos_C$ and $pos(a)$
 5: $pos_C \leftarrow pos(a)$
```

A pertinent aspect of the AB-protocol is the unit of communication between various clients. While prior work focusses on intention logs [5], in our protocol, we transmit the entire action (or a pointer to the stored procedure call). By this lazy replication of actions at other clients, we make the AB-protocol independent of the high abort rates often witnessed in optimistic methods of concurrency control.

The reconciliation procedure in our protocol, Algorithm 3, is designed to prevent the optimistic state from diverging too far from the stable state, by rolling back and re-applying optimistic actions when an actual conflict is discovered.

Algorithm 3: Reconciliation Protocol

```
1 Require: $Q$ a queue $[\langle a_1, v_1\rangle, \ldots, \langle a_k, v_k\rangle]$ of unreconciled optimistic
  evaluations
  1: CO($WS(Q)$) $\leftarrow$ CS($WS(Q)$)
  2: $Q \leftarrow []$
  3: for $j = 1; j <= k; j++$ do
  4:     Apply $a_i$ to CO producing result $v = a_i$(CO)
  5:     Insert $\langle a_i, v\rangle$ into $Q$
```

We use an approach proposed previously [52], which assumes that actions contain code to check for conflicts. When an action is re-applied, it either computes appropriate new result values or else it detects a fatal conflict and behaves as a no-op to simulate aborting.

Our action-based protocol has two advantages. First, it guarantees low latency because of one phase commit, while allowing any kind of interaction including object contention in the virtual environment. A second advantage is that the central server does not execute any actions, and therefore is independent of the application logic. The server only timestamps actions, queues them for delivery to clients, and manages the network traffic; this allows it to be highly scalable. This virtual timestamp, together with the positions of actions on the queue at the server, establishes virtual synchrony between the server and the clients [7]. Popular systems such as SIMNET [64, 9] and WAVES [30] use similar protocols at the object level — they broadcast updated data objects to all clients.

Correctness of our protocol is easy to establish. By the virtue of timestamping and ordering of actions on the server, each client executes every action that originates anywhere in the system, *in the same order*, on the same initial world state, CS. With our previous assumption of identical clients, every client would produce the same final state of the database. Action based protocols use active replication and therefore only need to use a one-phase commit protocol. Since replicas are executing transactions in parallel, the commit log of an action is the same for all replicas. Further, there is a fundamental determinism-concurrency tradeoff of the action based protocol. Serial execution of transactions can only be as fast the computational footprint of one replica. The only constraint of AB-protocol is deterministic execution of actions at client machines; and the designer is free to choose any platform that satisfies this constraint. [66] gives an novel model to introduce concurrency at clients by imposing a deterministic schedule on transactions across all replicas of a distributed system.

2.4 Minimality in AB-Protocols

The AB-protocol achieves a minimal transactional latency, but does not scale well. Trivially, if every client were to execute all actions for the entire system, each client would need resources of the order of the central server in a server-side architecture. Fundamentally, we would like to limit the computation at each client without forgoing the consistency and latency achieved by the AB-protocol. To achieve better scalability we explore minimality of computation in the AB-protocol, as described next.

2.4.1 Causality of actions

In the realm of object based protocols, numerous optimizations have been proposed to reduce the number of messages that are sent to each client [17, 24]. Most of these optimizations are variants of *area-of-interest paradigm* [25, 48]. In such models, the server restricts the set of update messages (and object data) to the visibility of a player in the virtual world. Although one could consider gen-

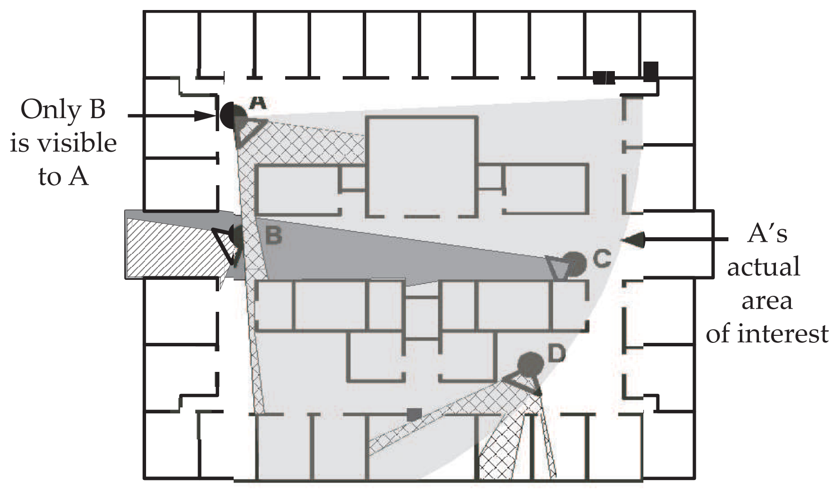


Figure 2.2: The RING system limits itself to the visibility of players, resulting in an inconsistent state across clients. The actually area that can causally influence *A* is much larger than its visibility.

eralizations of the methods proposed in such systems to action-based protocols, we next present an argument on why such an approach is not a general solution to the scalability problem.

A first observation is that restricted visibility applies only to movement-like actions and does not generalize well to arbitrary actions. For example, the RING architecture requires that the designer create an obstruction layer representing the objects blocking visibility. This obstruction layer is what is used to partition the database replicas [17]. If the game designer wants to base actions on other senses such as sound or scent, she must create a separate obstruction layer for each new sense. Furthermore, in cases like our example of a scrying spell from Section 2.1, there may be no obstruction information at all.

Furthermore, the usage of syntactic constraints such as restricted visibility has a deeper, subtle problem: the constraints are not sufficient to ensure consistency. For example, none of the current proposals cover transitivity of actions—characters can easily interact with one another even if they cannot see one an-

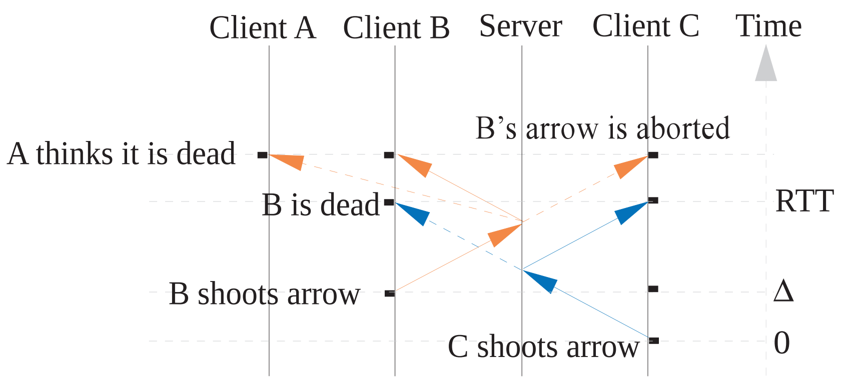


Figure 2.3: Inconsistency in area of interest paradigm

other. We illustrate this problem in Figure 2.2. Although players A, B, C, and D (filled circles) all inhabit the same virtual environment, very little interaction (filled and hatched polygons) is possible due to the occlusion of walls (solid lines). In fact, in this example, only two direct interactions are possible — between players A and B; and between players B and C. The restricted vision paradigm suggests that each action submitted by B would only affect A and C, whereas an action submitted by C would only affect B (because A cannot see C). However, this observation leads to an inconsistent state in the system as described next.

We illustrate the inconsistency using a scenario in a battlefield (Figure 2.3). In the following example, we denote the network latency from client machine to server as RTT, i.e. time it takes for a client machine to send a network packet to the server and get back an acknowledgement. Consider the following sequence of actions:

1. At time $t = 0$, C shoots an arrow at B.
2. At time $t = \Delta$, B shoots at A. We can assume $\Delta < RTT$, since otherwise the client machine of player B has received and executed the action of C, and so already knows that player B is dead.

3. At time $t = RTT$, machine of player B receives and executes the action of player C. B dies.
4. At time $t = RTT + \Delta$, client machine of player A gets the action of player B. A dies.

Ideally, in the above scenario, player B should die before it actually shot the arrow. However, the client machine of player B receives the action with C's shoot request only at time $t = RTT$, and by this time it has already sent player B's shoot request to the server. The client machine of player A receives B's shoot request at time $t = \Delta + RTT$, and subsequently determines that A is dead. It is interesting to note that player A could have determined B's death only if it also knew that C had shot B.

We conclude that although there is a bound on the visibility of a player, the actual area that can influence a player is much larger than the visible region (Figure 2.2). The primary limitation of prior work is that it assumed a *syntactic* restriction on influence of actions, however the influence is really determined by a causal relationship between actions in the virtual world. In the next section, we propose a protocol that leverages this causality.

2.4.2 MinAB-Protocol

We learned from the previous section that actions in virtual worlds directly affect only those objects that lie within a specific range (as determined by visibility). The information of objects within visibility range, however, is not enough to uniquely (and correctly) determine the outcome of actions. The causal dependence of actions depends on the nature of interactions and typically cannot be captured by syntactic constraints.

In this section we introduce the Minimal Action Based Protocol (MinAB-Protocol) that minimizes computational requirements of the clients. The protocol works by resolving the consistency problems that we discovered in the previous section.

In order to understand the causality of actions, let us begin by examining the transaction that is encapsulated in an action. We make an assumption that the read-set and write-set for an action are known a priori; as discussed in the previous section, the read and write sets of an action are limited to visibility in most virtual worlds and can therefore easily be determined. Next, we change the system architecture such that the central server also maintains an object set S, and although it does execute any actions, it updates this object set with the writes of actions executed by the clients. For simplicity of exposition, we denote by $W(S, v)$ an event that unconditionally stores the values v into the object set S.

Armed with these definitions, we can now change our protocols as shown in Algorithms 4, 5 and 6. To execute an action a, a client first executes a on `CO`. It simultaneously submits a to the server for serialization. Upon submission, the server first attaches a monotonically increasing sequence number to a. The server then finds a set of actions $A(a)$ such that each action $A(a)$ is serialized before a and if $a' \in A(a)$, then either of the two hold:

- $WS(a') \cap RS(a) \neq \phi$
- $a' \in A(a'')$ and $a'' \in A(a)$

Effectively, the server finds a set of actions that are necessary to determine the outcome of a. It sends the set $A(a)$ back to the client. The client executes these actions, in order, on `CS`. The results of applying locally originated actions to `CO` and `CS` are compared, and disagreements are reconciled if necessary.

Algorithm 4: MinAB Client-Side Protocol

```
1 Require: $Q$ a queue $[\langle a_1, v_1\rangle, \ldots, \langle a_k, v_k\rangle]$ of unreconciled optimistic
  evaluations
   1: Create action $a$ and apply to get $v = a(\texttt{CO})$
   2: Add $\langle a, v\rangle$ to $Q$ and send $a$ to server.
   3: Wait to receive action $b$ from the server.
   4: if $b$ is not an action in $Q$ then
   5:    {Either $b$ originated at another client or}
   6:    {is a blind write created by server}
   7:    Apply action $b$ to CS
   8:    for each write $x \leftarrow v$ performed by $b$ do
   9:       if $x \notin WS(Q)$ then
  10:          Perform the write $x \leftarrow v$ on CO
  11: else if $b = a_1$ then {$b$ must be head of $Q$}
  12:    Apply $a_1$ to CS to get result $u = a_1(\texttt{CS})$
  13:    if $u = v_1$ then {optimistic evaluation okay}
  14:       Remove $\langle a_1, v_1\rangle$ from $Q$
  15:    else
  16:       Reconcile CO with CS via Algorithm 3
  17: Send completion message $\langle a_i, u\rangle$ to server
```

The advantage of our protocol is that a client does not (necessarily) evaluate every action, only those that affect its transactions, thus saving both the execution time at the clients and the network bandwidth. In order to further optimize our proposal, we augment the client protocol to return a *completion message* when the stable result of an action is produced. The server uses these messages to construct S, an authoritative stable world state. The server performs analysis of read and write sets (Algorithm 6) to determine independently for each client which additional actions must be sent for evaluation because they (transitively) affect the client's submitted actions.

An interesting aspect of the MinAB-protocol is that it can be made tolerant of client failures at a reasonable cost in network bandwidth, by asking each client to send completion messages for *every* action it applies, not just its own. With this change, the only case in which the server does not receive a response to

Algorithm 5: MinAB Server-Side Protocol

```
1 Require: $\texttt{S}$ is the authoritative state
2 Require: $Q$ is a global queue of actions
3 Require: $sent(a)$ is set of clients sent action $a$
4 Require: $\texttt{S}(i)$ is state after applying actions $a_1 \dots a_i$. For the least $j$ such
  that no response for $a_{j+1}$ was received, the server holds $\texttt{S}(j)$ as well as
  $a_{j+1} \dots a_n$.
 1: Wait for message from client $C$
 2: if message is an action $a$ then
 3:    Timestamp $a$ and put it into $Q$
 4:    $pos(a) \leftarrow$ index of $a$ in $Q$
 5:    $sent(a) \leftarrow \emptyset$
 6:    Compute a reply to $a$ using Algorithm 6
 7: else if message is completion for $a_i$ then
 8:    Server holds message until $\texttt{S}(i-1)$ available
 9:    Installsinto $\texttt{S}$, resulting in $\texttt{S}(i)$
10:    Discard $a_i$ from action queue
```

Algorithm 6: Transitive Closure(A)

```
1 Require: $a_i, \dots, a_n$ is the action queue
2 Require: $a_{n+1}$ has just arrived from client $C$
3 Require: + denotes prepending action to sequence
 1: $A \leftarrow \{a_{n+1}\}$
 2: $S \leftarrow RS(a_{n+1})$
 3: for $j = n$ to $i+1$ do
 4:    if $WS(a_j) \cap S \neq \emptyset$ then
 5:       if $C \in sent(a_j)$ then
 6:          $S \leftarrow S \setminus WS(a_j)$
 7:       else
 8:          $S \leftarrow S \cup RS(a_j)$
 9:          $A \leftarrow a_j + A$
10:          $sent(a_j) \leftarrow sent(a_j) \cup \{C\}$
11: $A \leftarrow W(S, \texttt{S}(S)) + A$
12: return $A$
```

some action is when all clients that evaluate that action have failed. In such cases, it is acceptable to assume that the action was never submitted. The client can also be optimized for memory. The server can inform the client periodically of the last installed action, enabling the client to garbage collect the results of actions received in the past that it is no longer explicitly interested in. The correctness of our algorithm is stated as follows:

Theorem 2.4.1. *If the server follows Algorithm 5 and all clients follow Algorithm 4, then in a distributed snapshot of the system the states* `CS` *at the clients and the state* `S` *at the server will never be inconsistent.*

2.5 Boundedness of MinAB-Protocol

In the MinAB-protocol, each client evaluates only a necessary subset of the actions—those actions that actually affect the client. We investigate the boundedness of MinAB-protocol in this section.

To make the discussion more succinct and relevant to current MMOs, we assume that the virtual world follows the standard model of a discrete simulation engine, where the world state changes only at regular time intervals, the *simulation ticks* [70]. We denote the non-zero time interval between two consecutive ticks by τ.

Let us assume that a client could evaluate a set of actions AS in constant time γ such that γ is independent of the size of AS. Then the time for the server to receive a response for any action from a client should be $RTT + \gamma$, where RTT is the round-trip time between the client and the server. This implies that the server would need to send to the client a subset of actions that it has seen in the previous $(RTT + \gamma)/\tau$ ticks, which provides our first bound. In our analysis, we made an assumption that network latency is equal across all clients—we can easily drop this assumption by substituting RTT_{max} for RTT. Assuming that all clients have reasonable latency, and the virtual environment is very large, we believe that this is still a reasonable bound.

The bound is not valid in practice, however, because the time required for the client program to execute the set of actions AS is, in the worst case, proportional to the size of AS. We observe this when the time to execute the set of actions in A is of the order of RTT. In such a case, the time after which the server receives a response for an action translates into $2 \times RTT$. This increase in time consequently increases the size of the subsequent set of actions that is sent to the client. A trivial analysis of this phenomenon shows that the number of actions in A increases geometrically, thereby invalidating the previously obtained bound.

2.5.1 Hyperactive Replication Model

A drawback of MinAB-protocol is that when a client submits a new action after having been idle for a while, the server may respond with an unboundedly large set of actions. However, our MMO semantics provide us a limit on the size of this set. Most existing VEs have strict properties of locality that we can exploit. Every participant in can be represented as a high-dimensional tuple. Furthermore, this tuple has a finite maximum rate of change in position. For example, traditional spatial attributes like the position of a player cannot change more than the maximum object velocity. Similar restrictions apply to attributes like health if the virtual world has a maximum damage amount. As a result, many of the actions are restricted to a ball of fixed radius about a high dimensional point determined by the participant. For example, when a combatant is looking a target to attack, this is ball about the combatant's attack power and spatial position.

We use the MMO semantics to scale our system. In the situation described above, we know the position of the balls at time t and the maximum rate of change. If we couple this information with an action A of some other participant,

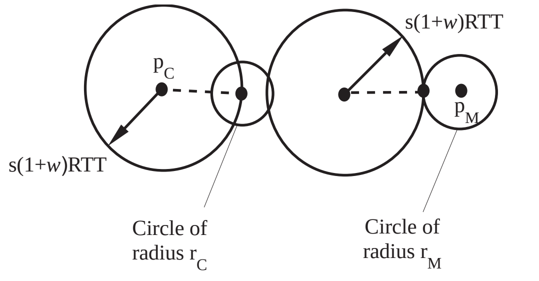


Figure 2.4: The worst-case in Hyperactive Replication Model

we can use simple geometrical calculations to answer the following question — can the participant's future actions be directly affected by the outcome of A?

The server now works as follows. It proactively pushes to each client a set of actions AS that *may* affect its future actions. The server therefore does not wait for a client to submit an action A. Such a push enables the client to execute the actions of AS during what would otherwise be idle time. In particular, at regular intervals of $\omega\, RTT$ time, where $0 < \omega < 1$, the server sends to each client all actions submitted in the previous $\omega\, RTT$ that could possibly affect any of future actions of the client.

Claim: The server receives a response for any action A from the client in time $(1 + \omega)\, RTT$ of sending A to the client.

Proof: We assume that it takes ½RTT time for an action to travel from the server to the client. Therefore, if an action A (along with some other actions) is sent to the client j ticks after the closest ω RTT cycle from the server, where $j \leq \omega\, RTT/\tau$, it reaches the client j ticks after the client has finished executing the previous action set. The client can therefore execute A in at most j ticks and respond back to the server. The response takes an additional ½RTT. Since j is bounded by $\omega\, RTT$, the maximum time for this entire process is $(1 + \omega)\, RTT$.

As stated earlier, the decision whether an action A is sent from the server to a client is based on whether or not the client's future actions could possibly conflict with A. Let the maximum area of influence of A in the virtual world be given by a sphere centered at the point $\bar{p}_A$ and radius r_A. Let the position of the character representing client C be given by $\bar{p}_C$, and let the maximum radius of influence of an action by C be r_C, and let the maximum rate of change in position of any object be given by s. Then A can affect any of C's future action in time $(1 + \omega)\, RTT$ if and only if

$$\| \bar{p}_A - \bar{p}_C \| \leq (2s \times (1 + \omega)\, RTT) + r_C + r_A \tag{2.1}$$

This equation reflects a worst-case in which A affects an object at distance r_A from itself, that object and C's character move towards one another, each traveling at maximum speed s, and they approach to distance r_C within the specified time bound of $(1 + \omega)\, RTT$, as illustrated in Figure 2.4. The equation gives us the first bound on the number of actions that can directly conflict with the actions of the client, represented as a sphere centered at the position of the client in the virtual world.

2.5.2 β-Hyperactive Replication Model

Though the Hyperactive Replication model gives a bound on the number of actions that can directly conflict with a client's actions and therefore have to be sent to the client, the actual set of actions that are sent to a client is the transitive closure of actions that conflict with the aforementioned set of actions.

We claim that the number of uncommitted actions than can directly or indirectly cause a conflict with any given action is unbounded. We illustrate this using the following example.

Dining Philosophers Problem. Consider a scenario with n participants, with each of them trying to grab two forks—one to their left and one to their right.

Let them be organized in the form of a circular ring located on earth's equator. If each of them tries to pick up the two forks at the same tick, then although the direct conflicts never involve more than two participants, a transitive closure of conflicts encompasses the entire world.

In order to counter this problem, we believe that the prevalent uncertainty in the system can be used to break the long chains. This can be used to restrict the size of the transitive closure of actions by *aborting* some actions a priori. A possible alternative to aborting actions is delaying actions by some amount of time so that the bulk of the actions in the conflicting action set are committed.

Determining the optimal way to abort actions is non-trivial. Issues such as fairness are prevalent—what if the actions for the same client are repeatedly aborted or delayed? Another issue is to find the minimal set of actions to abort in real-time, given the fact that most VEs are online and demand immediate response. With more and more people joining VEs, a fear in such a protocol is that the cost of evaluating transitive closures of conflicting actions might surpass the cost of processing actions at the server. Evaluating all such techniques is beyond the scope of this paper, and is interesting area for further research. algorithm to decide the fate of submitted actions.

As a first step towards solving this problem, we propose the β-Hyperactive Replication model. This model greedily decides whether or not an action should be aborted. Since all clients do not submit actions exactly at the same time, we believe that the random order of arrival of actions at the server will ensure fairness, i.e. the probability of an action getting aborted is the same for all clients. The greedy nature of the algorithm is computationally inexpensive, and therefore we conjecture that the model can be used in real-time environments.

Algorithm 7 shows the modules of the β-Hyperactive Replication model.

Algorithm 7: β-Hyperactive Replication Model

```
1 Require: Variables actionCount, previousCount, lastCommitted, and
  numClients are global

2 function onActionSubmission(action) {
    1: $A_{actionCount} \leftarrow action$
    2: $i \leftarrow actionCount$
    3: for ($j = 0$ to $clientCount - 1$) do
    4:     if $|p_{A_i} - p_{C_j}| \leq (2s \times (1+\omega)\ RTT) + r_C + r_A$ then
    5:         $clientConflicts_{i,clientConflictCount_i} \leftarrow j$
    6:         $clientConflictCount_i$ += 1
    7: $actionCount$ += 1
  }

  function onNextTick() {
    1: for ($i = previousCount$ to $actionCount - 1$) do
    2:     $S \leftarrow RS(A_i)$
    3:     $invalid \leftarrow$ false
    4:     for ($j = i - 1$ to $lastCommitted + 1$) do
    5:         if $isValid_j$ and $S \cap WS(A_j) \neq \emptyset$ then
    6:             if $|p_{A_i} - p_{A_j}| >$ threshold then
    7:                 $invalid \leftarrow$ true
    8:                 break
    9:             $S \leftarrow (S - WS(A_j)) \cup RS(A_j)$
    10:            $conflicts_{i,conflictCount_i} \leftarrow j$
    11:            $conflictCount_i$ += 1
    12:    $isValid_i \leftarrow$ not $invalid$
    13: $previousCount \leftarrow actionCount$
  }
```

The function `onActionSubmission()` is called when any client submits an action. This action is added to a global queue of actions (line 4). The function then evaluates the set of clients (given by *clientConflicts*) that could be interested in the action sometime in the near future. The second function `onNextTick()` is invoked at every tick, i.e. at regular intervals of time τ. The identifier range [*previousCount, actionCount*) gives the identifiers of all actions submitted in the previous tick. For each submitted action A, `onNextTick()` evaluates into *con-*

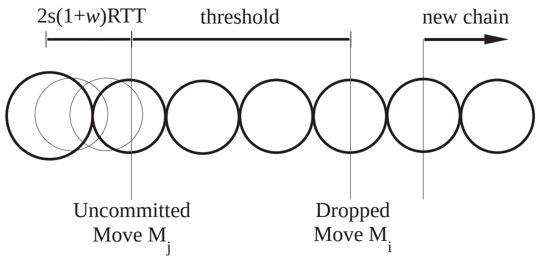


Figure 2.5: Chain breaking in the *beta*-Hyperactive Replication Model

flicts a transitive closure of all conflicting uncommitted actions. If any of the conflicting actions is at a distance greater than some *threshold* distance from A, then A is aborted.

The Hyperactive Replication model and the β-Hyperactive Replication model together give two bounds. The first bound is on the maximum number of actions that need to be sent to a client due to direct conflicts, represented as a function of time and distance in the attribute hyperspace. The second bound is on the maximum number of actions that can be a part of any actions transitive closure, represented as a function of distance. Combining these two bounds, we get the following (loose) bound on the number of actions sent to a client at each tick, represented as a function of time and distance:

$$\| \bar{p}_A - \bar{p}_C \| \leq (2s \times (1+\omega)\, RTT) + r_C + r_A + \mathit{threshold} \tag{2.2}$$

An important aspect of the Bounded Hyperactive Replication model is the conflict detection algorithm. Typically, virtual worlds require an unordered evaluation of actions with the same timestamp [71]. However, the decision to abort actions in our β-Hyperactive Replication model is sequential (lines 19-34). This enables the model to accept a majority of the actions, while aborting only those actions that invalidate the bound. To put things in perspective, we again consider the Dining Philosophers problem. If all participants try to pick up the two forks at the same tick, we conjecture that the decision to abort all of the requests is suboptimal. The primary reason for this is the fact that the intention was to break long chains, and not make a decision. By aborting a few actions at regular intervals, the chain can be broken into numerous pieces, each of which satisfies the requisite threshold.

2.5.3 Other optimizations

In the remainder of this section, we give two optimizations for our models. Though most of these techniques are popular in the graphics community [51], they generalize to the domain of event propagation. In particular, as we represent the virtual world by a high-dimensional database, we can apply many of these techniques to higher dimensions.

Inconsequential Action Elimination

Throughout the discussion in this paper, we have assumed that an action submitted by any participant can affect the future actions of all other participants that satisfy a certain bound on the distance between their positions. We claim that the number of such conflicts can be sharply reduced by integrating nontrivial MMO semantics into the system. For example, suppose that a virtual environment contains humans and insects. A participant who is pretending to be an insect in the VE would probably need to consistently know the location of other insects and of the humans. However, a participant who is acting as a human in the VE may not need to reliably know the locations of all of the insects. We can therefore extend the system so as to allow the clients to specify exactly what kind of actions and information they are interested in, instead of assuming absolute uniformity.

Area Culling

Another assumption that has been made is that the area of influence of any action is a sphere centered at its point of occurrence. However, most of the actions such as shooting an arrow, or even walking, normally have a velocity vector associated with them. Even health may have an associated "velocity" vector to it, if the damage is occurring over time. We can therefore integrate this velocity vector in the bound calculation to predict any future conflicts. The conflict equation (Equation 2.1) can be restructured as:

$$\| \bar{p}_M + (\bar{v}_M \times (t_M - t_C)) - \bar{p}_C \| \leq (2s \times (1 + \omega)RTT) + r_C, \tag{2.3}$$

where $\bar{v}_M$ is the velocity vector associated with M, t_M is the time of occurrence of M, and t_C is the time at which the position of client C was last updated to $\bar{p}_C$. Note that the term, r_M, corresponding to the area of influence of M is now represented as a vector and moved to the left hand side of the equation.

2.6 Experiments

To examine the performance characteristics of action based protocols, we built a virtual world that used the MinAB-protocol in Java 5.0 and conducted experimental studies to quantify and evaluate its performance. We call our implementation *SEVE*, for Scalable Engine for Virtual Environments. We also built a virtual world that used server-side architecture. We put in our best effort to create an optimized system to represent current online virtual worlds such as Second Life or World of Warcraft. Furthermore, we implemented the NPSNET and the RING architectures, which represent the state of the art in distributed simulations.

Our experimental evaluation is based on a synthetic workload that stresses the consistency issues in MMOs. We generated the synthetic workload for a simple virtual world, similar to the example in Section 2.4.1. We call this virtual world *Manhattan People*. It consists of avatars moving about in a rectangular area and colliding with walls or other avatars. Whenever an avatar bumps into something, it changes its direction by 90°. By adjusting the number of walls, we controlled the computational complexity per action, while we controlled the expected number of conflicts between actions by varying the number of participants.

2.6.1 Experimental Setup

System Setup

We obtained performance results by running the virtual world on an EMULab [69] testbed consisting of 65 machines—64 clients and 1 server. Each EMULab machine was a Pentium III Processor with 2 GB of RAM, running Linux 2.4.0. We report the timings obtained using the Java `System.currentTimeMillis()` method. Each machine, except one designated as the central server, was running other programs such as a desktop manager, a document editor and a web browser in the background. We consider this a simple way to emulate a typical client machine. Additionally, we used EMULab to introduce latency at the network level in order to simulate deployment on a wide-area network. The average latency between machines was 238ms. The numbers we present are repeatable, and were averaged over 10 runs of the system, with each run lasting approximately 1 hour.

Table 2.1: Simulation Settings

Virtual world size	1000 x 1000
Number of walls	0 – 100,000
Number of clients	0 – 64
Average latency	238ms
Maximum bandwidth	100Kbps
Moves per client	100
Move generation rate	Every 300ms per client
Move effect range	10units
Avatar visibility	30units
Threshold	1.5 × Avatar visibility

Virtual World Setup

We fixed the size of the virtual environment in Manhattan People at 1000 x 1000 points. Each wall had length 10, and the number of walls was limited to 100,000. Each action checked for conflicts with a varying number of walls closest to the client's avatar, and all other avatars within walk-able range. Checking for collisions with walls, we made heavy use of trigonometric functions—a complexity that was forced in to simulate the performance of virtual worlds such as Second Life.

Our simulations showed that the average time required to execute a single action is linear in the number of walls in the virtual world. In our system, we noticed that systems used an average of 6.95ms to execute an action, per 1,000 visible walls (1,000 is very close to the average number of walls a client sees for 100,000 walls in our virtual world). Table 2.1 gives an overview of the simulation parameters.

2.6.2 Performance Evaluation

We performed three batteries of experiments. First, we evaluated the scalability-complexity tradeoff in (a) a server-side model (Central)—to represent Second

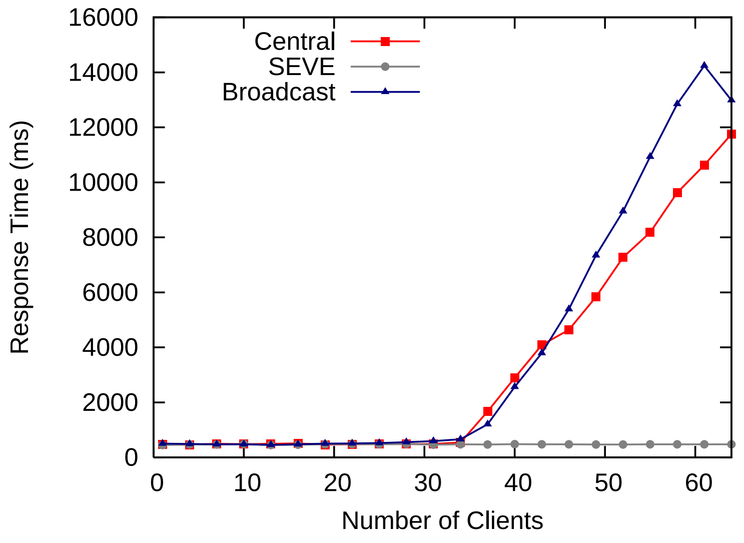


Figure 2.6: Scalability of SEVE vs. Central architecture

Life and WoW, the state of the art in online games; (b) a broadcast model (Broadcast)—representing NPSNET and SIMNET, the state of the art in distributed simulations; and (c) our action based distributed model (SEVE). Second, we explored the bandwidth requirements of the three models. Third and last, we evaluated the consistency-performance tradeoff.

Scalability vs. Complexity

For this first set of experiments, every single client submitted a total of 100 actions at intervals of 300ms per action. The number of walls was fixed at 100,000, while we varied the number of clients between 0 and 64. In a single run of the simulation, the number of other avatars that a client's avatar could see was empirically determined to be 6.87 on average.

We empirically determined that the time it took for a machine to evaluate a single action was 7.44ms. Figure 2.6 compares the response time observed by clients against the number of clients. As apparent from the figure, the server-side architecture and the broadcast model break down at about 30-32 clients. This is not too surprising for the server-side architecture since for every action that a client submits, the server has about 300ms to evaluate it. If 32 clients submit actions simultaneously, each action consuming 7.44ms of a server's time, the total time required to evaluate a round of actions is 240ms. The remaining 60ms can be attributed to synchronization and networking overhead. As noted earlier, each client in the broadcast model has computational requirements comparable to the central server; and therefore we observe a similar scalability for the broadcast model.

In contrast to that, SEVE's response time remained perfectly stable as the number of clients increased. We empirically determined the time for calculating the transitive closure of conflicts over a single action to be 0.04ms on average. However, as the number of clients goes up, so does the number of concurrent actions and the time required to evaluate a transitive closure. This factor is alleviated by the fact that the size of the transitive closure is bounded as a result of the actions getting aborted. We performed experiments on a single server and determined the limit of our implementation to be about 3500 clients.

Figure 2.7 compares the response time observed by the clients against the time it took to evaluate a single action. The number of clients employed in this experiment was fixed at 25. The server-side model and broadcast model performed well for actions that took less than 10ms for processing. However, as the complexity increased, the response time increased drastically, effectively making the game unplayable. Again, the response time for SEVE remained unaffected.

Finally, we evaluated the sensibility of SEVE with respect to the density of avatars. Recall that humans are social beings, so avatars can be expected to

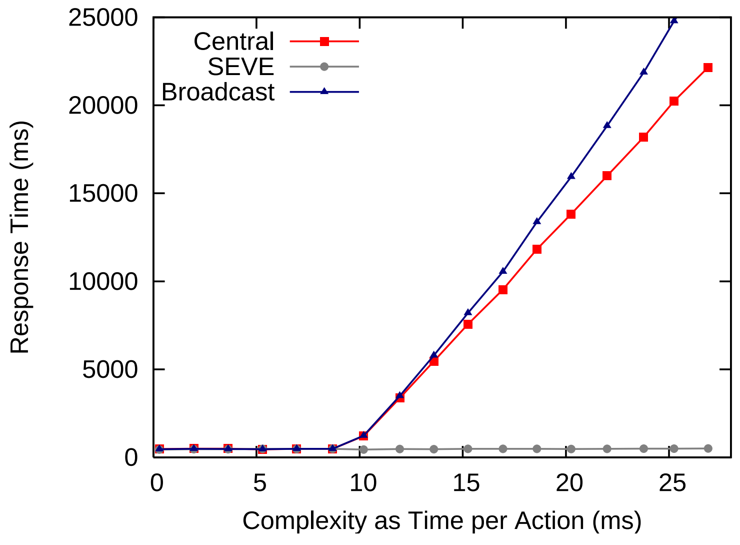


Figure 2.7: Response Time vs. Action Complexity

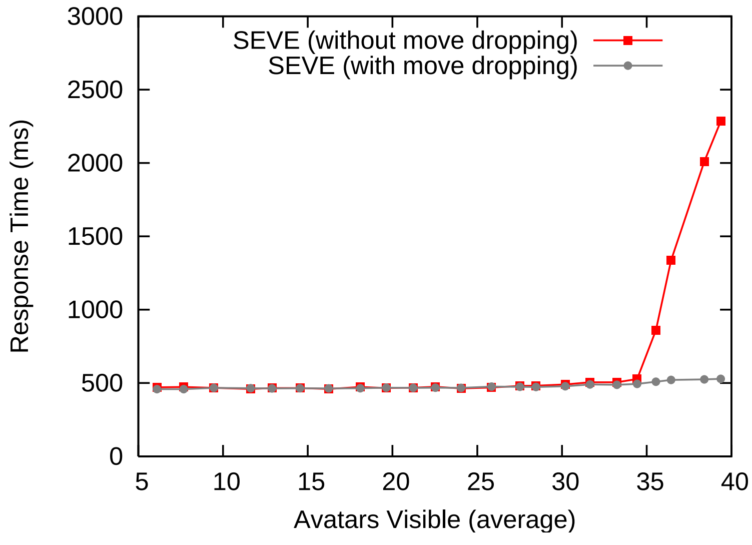


Figure 2.8: Effect of increasing density of avatars

form clusters in a real system. For this test, the number of clients was fixed at 60. The size of the virtual world was reduced to 250x250 units, and the avatars were initially positioned 4 units apart from each other. We varied the visibility of avatars from 10units to 100units. Figure 2.8 gives the observed response time versus the average number of other avatars visible to each avatar.

Action effect range	1	3	5	7	9	11
% Actions aborted	0	0	0.01	1.53	4.03	8.87

Table 2.2: Percentage of actions aborted (visibility = 20units)

The naive implementation of SEVE bogged down as the number of visible avatars exceeded 35, primarily because the clients ran out of computational power. In comparison, the improved implementation of SEVE started aborting actions that were causing long chains, allowing it to keep response time stable regardless of the density of avatars. The number of aborted actions varied from 1.5%-7.5% for different runs of the system.

At this point, it should be noted that the percentage of actions aborted is in fact independent of avatar visibility. This is because the length of chains depends on the *range of action effect*, and not avatar visibility. Table 2.2 gives the percentage of actions aborted as a function of action effect range. While the numbers appear to be fairly high for a large action effect range, the density of avatars in this particular experiment is really an extreme case. We can safely consider this a worst case scenario.

Varying the number of actions per client, or the rate of action generation had no impact on the performance of SEVE. The server-side model and the broadcast model, however, diverged when the number of actions, or the rate of generation, was increased. We omit the corresponding graphs due to space constraints.

Bandwidth Requirements

A main concern of distributed systems is in the amount of network traffic generated. Figure 2.9 shows the comparison between Central, Broadcast and SEVE. As expected, the broadcast model requires excessive network traffic (quadratic in the number of clients). This was the original reason why systems such as

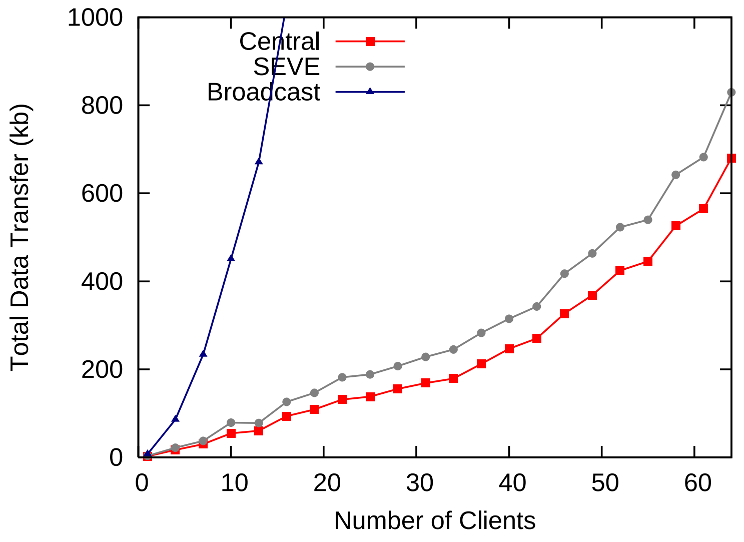


Figure 2.9: Total data transfer

RING were proposed. We note that the total traffic for the server in SEVE does not differ significantly from a server-side model, which obviously is optimal in total traffic. We conclude that SEVE does not incur higher costs on network infrastructure than current systems.

Performance vs. Consistency

We evaluated the performance impact of calculating transitive closures in SEVE with 64 clients and 100,000 walls compared to a RING-like architecture which only evaluates actions within the visible range of an avatar. The average number of avatars that an avatar could see was increased to 14.01 as opposed to 6.87 earlier, leading to more conflicts processing at the clients. Figure 2.10 shows the results we obtained.

Calculating the transitive closure in SEVE accounted for a runtime overhead of 1% compared to the RING-like architecture. This shows that the runtime overhead of our strongly consistent approach is negligible.

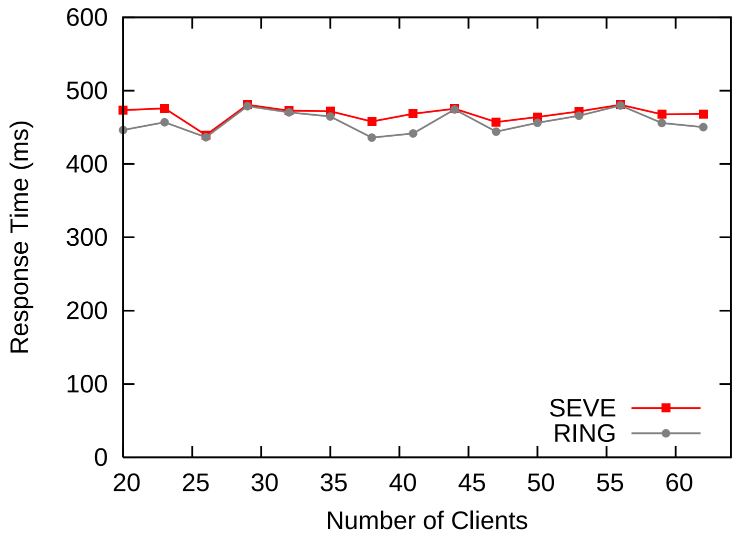


Figure 2.10: SEVE vs RING-like Architecture

In summary, our experiments show that our architecture is massively scalable while preserving strong consistency. It gives an order of magnitude improvement over existing strongly consistent architectures for networked virtual environments.

2.7 Conclusions

In this paper we motivate that at the core of networked virtual environments lie data management problems. We identified an interesting concurrency problem to which we proposed a novel practical solution based on taking semantics into account. We believe, however, that we just scratched the surface of this (for the database community) new area, and that both virtual worlds as well as other virtual networked environments — from collaborative problem solving to online games — can benefit from solutions from the database community for years to come.

CHAPTER 3

ENTANGLED QUERIES

3.1 Introduction

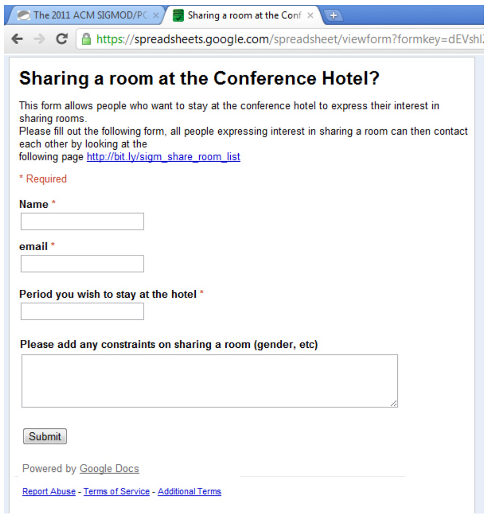
3.1.1 Declarative data-driven coordination

Collaboration and coordination are increasingly important aspects of the ways people produce, process and consume data. This is true not only for serious tasks such as scientific dataset management, but also at the grass-roots level, as internet users organize and coordinate activities online. In [34], the authors presented the vision of *declarative data-driven coordination* (D3C) as a high-level design principle for collaborative data management systems. In this paper, we address some of the challenges related to making D3C a reality by introducing a system that supports *entangled queries* – a declarative mechanism for data-driven coordination.

The paper [34] motivates D3C through a series of real-world coordination scenarios. We revisit these examples here and explain D3C in some detail in order to make more concrete the technical challenges involved in implementing entangled queries.

A common coordination scenario is joint travel planning with friends or family; for instance, several colleagues on a business tour might wish to separately reserve rooms at the same hotel. The desired coordination is based on attributes of the data itself, such as hotel name and date, rather than on context information such as the time the booking is made. Thus, the coordination itself is *data-driven*.

An example of such coordination that we witnessed recently was a room


The 2011 ACM SIGMOD/PC
Sharing a room at the Conf
https://spreadsheets.google.com/spreadsheet/viewform?formkey=dEVshl

Sharing a room at the Conference Hotel?

This form allows people who want to stay at the conference hotel to express their interest in sharing rooms.
Please fill out the following form, all people expressing interest in sharing a room can then contact each other by looking at the
following page http://bit.ly/sigm_share_room_list

* Required

Name *

email *

Period you wish to stay at the hotel *

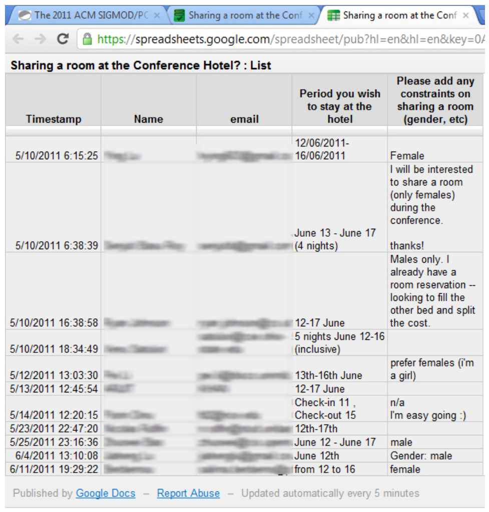
Please add any constraints on sharing a room (gender, etc)

Submit

Powered by Google Docs

Report Abuse - Terms of Service - Additional Terms

(a)


The 2011 ACM SIGMOD/PC
Sharing a room at the Conf
Sharing a room at the Conf
https://spreadsheets.google.com/spreadsheet/pub?hl=en&hl=en&key=0A

Sharing a room at the Conference Hotel? : List

Timestamp	Name	email	Period you wish to stay at the hotel	Please add any constraints on sharing a room (gender, etc)
5/10/2011 6:15:25	[redacted]	[redacted]	12/06/2011-16/06/2011	Female
5/10/2011 6:38:39	[redacted]	[redacted]	June 13 - June 17 (4 nights)	I will be interested to share a room (only females) during the conference. thanks!
5/10/2011 16:38:58	[redacted]	[redacted]	12-17 June	Males only. I already have a room reservation -- looking to fill the other bed and split the cost.
5/10/2011 18:34:49	[redacted]	[redacted]	5 nights June 12-16 (inclusive)	
5/12/2011 13:03:30	[redacted]	[redacted]	13th-16th June	prefer females (i'm a girl)
5/13/2011 12:45:54	[redacted]	[redacted]	12-17 June	
5/14/2011 12:20:15	[redacted]	[redacted]	Check-in 11 , Check-out 15	n/a I'm easy going :)
5/23/2011 22:47:20	[redacted]	[redacted]	12th-17th	
5/25/2011 23:16:36	[redacted]	[redacted]	June 12 - June 17	male
6/4/2011 13:10:08	[redacted]	[redacted]	June 12th	Gender: male
6/11/2011 19:29:22	[redacted]	[redacted]	from 12 to 16	female

Published by Google Docs – Report Abuse – Updated automatically every 5 minutes

(b)

Figure 3.1: SIGMOD Room Sharing Website

sharing website deployed for a conference (Figure 3.1). The SIGMOD 2011 room sharing tool was created using Google Docs as a simple form that is used to input data into a spreadsheet. A user who wishes to find a partner to share a room at the conference hotel expresses her intention by inputing her name, dates of travel, and constraints that she is bound by. On entering this information, the user is redirected to the spreadsheet, where she queries the data to see if there is an intention by another user who satisfies her constraints. On finding satisfactory partners, users communicate over email or phone to coordinate on hotel name and then proceed to make bookings.

There are many other settings in which users wish to coordinate. College students want to enroll in the same courses as their friends, busy professionals want to schedule joint meetings, and wedding guests want to purchase gifts in a way that avoids duplication. Coordination also occurs in massively multi-player online (MMO) games, where players are often interested in developing joint strategies with other players to achieve common objectives. Again, the coordination is data-driven as it relates to in-game goals.

Despite the ubiquity of scenarios such as those described above, coordination is not commonly supported in today's data-driven applications. For example, joint travel planning usually starts with significant out-of-band communication to fix an itinerary; this requires the use of email, telephone, or perhaps a more elaborate custom solution like the SIGMOD 2011 room sharing website discussed above. Next, one designated user makes a group booking, or all users try to make bookings simultaneously and hope that enough seats will remain available. Finally, more communication may be necessary to sort out finances. The same is true for the other examples of coordination mentioned above. In MMO games, for instance, joint strategies are currently formed using out-of-band communication, to the detriment of gameplay experience.

The idea behind D3C is to provide a way for users to coordinate within the system and without having to worry about the details of the coordination. Because the coordination is data-driven, the coordination abstraction is designed to sit at the same level as other abstractions that relate to the data. Declarativity – allowing users to express what is to be achieved, rather than how it is to be achieved – has long been an underlying design principle in databases. In a declarative specification of coordination, the users' only responsibility is to state their individual preferences and constraints, and the system takes care of the rest. D3C is thus in contrast with existing work on data-driven coordination in workflows [2, 47] and Web services [12, 26, 58], which does not clearly separate the coordination specification and mechanism.

To see what coordination looks like in a system that supports D3C, consider an example. Suppose Ron wants to travel to Paris on the same flight as Harry. In our system, he can express his request with the following *entangled query*:

```
SELECT 'Ron', fno INTO ANSWER Reservation
```

```
WHERE
fno IN (SELECT fno FROM Flights WHERE dest='Paris')
AND ('Harry', fno) IN ANSWER Reservation
CHOOSE 1
```

Harry also wants to travel with Ron, but he has an additional constraint: he wants to travel only on flights operated by United. His query is as follows:

```
SELECT 'Harry', fno INTO ANSWER Reservation
WHERE
fno IN (SELECT fno FROM Flights F, Airlines A WHERE
                   F.dest='Paris' AND F.fno = A.fno
                   AND A.airline = 'United' )
AND ('Ron', fno) IN ANSWER Reservation
CHOOSE 1
```

Section 3.2 explains the syntax of these queries in detail. For now, it is enough to understand that `Reservation` is a name for a virtual relation that contains the answers to all the current queries in the system. The `SELECT` clause specifies Ron's own expected answer, or, in other words, his contribution to the answer relation `Reservation`. This contribution, however, is conditional on two requirements, which are given in the `WHERE` clause. First, the flight number in question must correspond to a flight to Paris. Second, the answer relation must also contain a tuple with the same flight number but `Harry` as the traveler name. Harry's query places a near-symmetric constraint on `Reservation`.

Neither user explicitly specifies which other queries he wishes to coordinate with – e.g. by using an identifier for the coordination partner's query. Instead, the coordination partner is designated implicitly using the partner's query re-

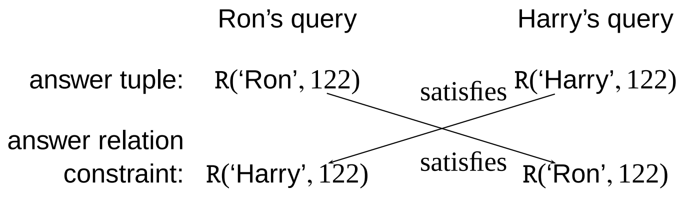
Flights

fno	dest
122	Paris
123	Paris
134	Paris
136	Rome

Airlines

fno	airlines
122	United
123	United
134	Lufthansa
136	Alitalia

(a)



(b)

Figure 3.2: (a) Flight database (b) Mutual constraint satisfaction

sult. This is a deliberate choice that allows coordination with potentially unknown partners based purely on desired shared outcomes. In travel planning, of course, it typically *is* known who one's coordination partners will be. However in other scenarios such as MMO games, coordination partners may be unknown and their identities irrelevant.

When the system receives Ron and Harry's queries, it answers both of them simultaneously in a way that ensures a coordinated flight number choice. In general, there may be many different suitable flights, but Ron and Harry only want to make a booking on one of them. The `CHOOSE 1` clause present in both queries specifies that only one tuple is to be returned per query. The tuples returned must be such that all constraints are satisfied. If the database is as shown in Figure 3.2 (a), the system non-deterministically chooses either flight 122 or 123 and returns appropriate answer tuples. Figure 3.2 (b) shows the mutual constraint satisfaction that takes place in answering for 122. The intent is that Ron and Harry should now be able to make a booking on flight 122.

The above queries are of course simplified to illustrate the basic coordination mechanic; in a real travel reservation setting, they would include checks for seat availability and other factors.

3.1.2 Enabling D3C

Existing related abstractions

Other research communities have long recognized the need for communication among concurrently running processes and have designed solutions to support it. Systems researchers have developed solutions ranging from low-level mechanisms such as message passing, shared memory, locks and semaphores to higher level abstractions such as transactional memory [39]. The programming languages community has given us Concurrent ML [56], Erlang [68], Stackless Python [55], Concurrent Haskell [27] and many other languages that come with concurrency support. These languages enable communication through channels or other mechanisms in a clean and precisely specified way. At a higher level, abstractions such the π-calculus [46] allow formal modeling and reasoning about communication.

The data management research community has long avoided the coordination problem, probably as a consequence of accepting isolation among transactions as a dogma. However, as pointed out above, data-driven coordination has real uses. The process-centric abstractions mentioned above are not a good fit for data-driven applications [34]; a large class of such applications would be much easier and faster to develop using a data-centric abstraction such as entangled queries. Moreover, a well engineered high-level abstraction like entangled queries creates an opportunity for automatically optimizing coordination on a large scale that is not possible for the lower-level abstractions offered by operating systems.

It is important to emphasize that existing database mechanisms such as nested transactions [44], Sagas [18], or ConTract [57] that weaken isolation in a form or another do not solve the coordination problem, for two reasons. First, they only allow for unidirectional information flow between transactions on the same conceptual layer (of nesting), not the kind of bi- or multidirectional flow required to achieve coordination. Moreover, coordination requires *automated matchmaking* between queries, a challenge which the work cited above does not address.

In fact, one may be biased towards mechanisms such as Cooperative Transaction Hierarchies [50] or Split Transactions [54] for enabling coordination. Both these mechanisms, with their specific application domain, require explicit declaration of the coordination structure. They are basic extensions of a system implementing shared memory for transactions and require declaration of data that is potentially either immutable or dependent on other transactions. This not only makes it very hard for the programmer to determine an execution, but also takes way a lot of flexibility from target applications.

Triggers or other active database constructs [72] may also seem relevant and appear to address the same problems as D3C, since active databases perform actions based on certain conditions becoming true in the database. However, trigger conditions are preconditions, while the coordination constraints of entangled queries are postconditions on the desired state of the database after the coordination. Again, triggers provide no straightforward way to achieve coordination matchmaking, which is the key problem addressed and solved in this paper.

Making coordination possible

Once the new entangled query abtraction has been formalized, a key technical challenge is to solve the coordination problem. That is, we need an algorithm that finds answers to the entangled queries in a way that satisfies the coordination constraints.

There is, however, a fundamental obstacle. The combination of a declarative query language such as SQL with coordination constraints of the kind illustrated above naturally captures the general Constraint Satisfaction Problem (CSP) of AI [15], which is NP-complete. This source of complexity is included by design: the very idea of D3C calls for a coordination solution to be a *choice* (nondeterministic, if you will) from a query result, *constrained by cross-query conditions*. Declarativity naturally entails a (combinatorial) satisfiability problem.

There are in fact two sources of nondeterminism (disjunction) and thus complexity in the coordination problem. The first is the choice of queries to be grouped together; the second, the choice of data tuples from the query results that are chosen as coordinating solutions. We cannot reasonably hope to eliminate the second type of complexity; this is the same issue that causes select-project-join queries to be NP-complete if one considers the query to be part of the input. On the other hand, one usually considers this acceptable because queries are small. If this second source of NP-completeness had to be eliminated, one could not support declarative queries with coordination constraints in a similar formalism.

A key contribution of this paper is a syntactic condition, *safety*, which ensures that coordination can be performed efficiently in the sense that the first source of complexity is eliminated. Coordination is only NP-hard in the size of the groups of queries or individuals who want to coordinate; in a travel scenario like our example where an arbitrary number of pairs of two people want to coordinate, this size is two. The hardness result is independent of the total number of entangled queries in the system, and also of the size of the data in the database. The latter fact is comfortingly obvious from the fact that the algorithm presented in this paper merges queries to be coordinated statically into standard SQL queries that only produce coordinated solution tuples for the constituent entangled queries; the essential query matching/coordination problem is solved without access to the data.

3.1.3 Contributions

The contributions of this paper are as follows. First, we formalize entangled queries, a simple yet powerful abstraction for D3C. Entangled queries are expressed in an extension of SQL, allowing the coordination constraints and the data involved in the coordination to be specified at the same level of abstraction. They are inspired by a language example from [34]; however, in this paper we give a full formal treatment of these queries, including a precise syntax and semantics.

Second, we introduce a formal notion of *safety* for queries that are admitted into the system. In keeping with our previous discussion, safe queries are designed to allow efficient evaluation in realistic settings rather than express generic CSP instances.

Third, we present an algorithm for coordination. The algorithm begins by working at the syntactic level to solve the *query matching* problem – identifying the potential coordination partners for each query. Next, each set of matching queries is combined into a larger query that expresses the desired joint outcome. For example, Harry and Ron's queries would be combined into a single query asking for a United flight to Paris. Finally, the answers to the combined query are used to generate individual answers.

Fourth, we introduce an end-to-end system that supports entangled queries. Apart from an optimized implementation of the algorithm, we present other components for query management and interaction with the application layer. Our system supports coordination in two modes: set-at-a-time mode (queries arrive in batches) and incremental mode (queries arrive as a stream). We leverage the properties of coordination structures to partition and evaluate query sets independently and in parallel.

Finally, we give experimental results that use our system and demonstrate the scalability of the coordination algorithm. We strive to use workloads that are as realistic as possible; in generating them, we make use of real social network data and extend them to a scale which is realistic for today's internet.

The remainder of this paper is organized as follows. Section 3.2 introduces the syntax and semantics of entangled queries. Section 3.3 discusses the kinds of coordination structure that are likely to be present in the most common use cases. Section 3.4 presents the evaluation algorithm for coordination. Section 3.5 and 3.6 describe our system implementation and contain experimental results, while Section 3.7 discusses future work. We mention related work throughout.

3.2 Entangled queries

In this section, we introduce a SQL-like syntax for entangled queries, propose an intermediate representation for ease of exposition and define the semantics of query answering.

3.2.1 Syntax

An entangled query is expressed in extended SQL using the following syntax:

```
SELECT select_expr
INTO ANSWER tbl_name [, ANSWER tbl_name] ...
[WHERE where_answer_condition]
CHOOSE 1
```

The `WHERE` clause is a normal condition clause that may refer to both database and `ANSWER` tables. The `ANSWER` tables are not normal database relations, whether permanent or temporary. Their purpose in the query is only to serve as names that are shared among queries and permit coordination. For example, the relation `Reservation` in the example from the introduction is an `ANSWER` relation. There is no relation named `Reservation` in the database; after the queries are evaluated, Ron and Harry each receive a result set with the appropriate answer tuple. These answer tuples do not persist anywhere, nor are they accessible to any other queries. In particular, Ron's answer tuples are not even accessible to Harry's query and vice versa. The `CHOOSE 1` at the end of the query explicitly specifies that the system should choose exactly 1 tuple among all the tuples which satisfy the coordination constraints, and that such a query should be chosen at random.

This paper presents semantics and an evaluation algorithm for entangled queries that are restricted to use only select-project-join (conjunctive) queries on the `ANSWER` relations in the `WHERE` clause, and arbitrary queries otherwise. Such queries are powerful and expressive enough to handle many real-world coordination scenarios. We discuss potential extensions in Section 3.7.

3.2.2 Intermediate representation

Although entangled queries are specified in an extension of SQL, their evaluation is easier to perform on an intermediate representation. The representation uses a Datalog-like syntax; however, it does not involve any recursion and it is completely equivalent to the SQL syntax presented above.

In this representation, an entangled query has the form

$$\{C\}\ H\ :\!-\ B$$

where C and H are conjunctions of relational atoms over answer relations and B a query over database (non-answer) relations. B, H and C are the *body*, *head* and *postcondition* of the query, respectively. Each atom in the representation may contain constants and variables. All variables that appear in H or C must also appear in B (a range-restriction requirement). For simplicity of discussion, we restrict B to conjunctions of relational atoms for the remainder of this paper. This is, however, not enforced by the model in general.

For an entangled query expressed in extended SQL, H corresponds to the `SELECT INTO` clause, while B and C correspond to information in the `WHERE` clause. C specifies all the conditions on `ANSWER` relations from the `WHERE` clause. B specifies the conditions on database relations from the `WHERE` clause, as well as serving to bind variables used in H and C.

Figure 3.3 (a) shows the intermediate representation of Ron and Harry's queries from the introduction. The relations `Reservation`, `Flights` and `Airlines` are abbreviated as `R`, `F` and `A` respectively.

$$\{\texttt{R}(\texttt{Harry}, x)\}\ \texttt{R}(\texttt{Ron}, x)\ :\!-\ \texttt{F}(x, \texttt{Paris})$$
$$\{\texttt{R}(\texttt{Ron}, y)\}\ \texttt{R}(\texttt{Harry}, y)\ :\!-\ \texttt{F}(y, \texttt{Paris}) \wedge \texttt{A}(y, \texttt{United})$$

(a)

1:	$\{\texttt{R}(\texttt{Harry}, 122)\}$	$\texttt{R}(\texttt{Ron}, 122)$
2:	$\{\texttt{R}(\texttt{Harry}, 123)\}$	$\texttt{R}(\texttt{Ron}, 123)$
3:	$\{\texttt{R}(\texttt{Harry}, 134)\}$	$\texttt{R}(\texttt{Ron}, 134)$
4:	$\{\texttt{R}(\texttt{Ron}, 122)\}$	$\texttt{R}(\texttt{Harry}, 122)$
5:	$\{\texttt{R}(\texttt{Ron}, 123)\}$	$\texttt{R}(\texttt{Harry}, 123)$

(b)

Figure 3.3: (a) Intermediate representation of entangled queries (b) Grounded queries

3.2.3 Semantics

From the point of view of a single entangled query, evaluation is a process that returns an *answer*, i.e. a single row from the appropriate answer relation. From the point of view of the system, evaluation always involves a set of entangled queries, and the goal is to *populate* the answer relation in a way that respects all queries' coordination constraints. In the running example, Ron and Harry wish to coordinate on flight numbers. The system evaluates their queries by finding a tuple for Ron's query and a tuple for Harry's query that share the same flight number, and returning each tuple as an answer to the appropriate query.

Consequently, coordination semantics must be defined from the perspective of the system, by specifying how a set of entangled queries must be answered together. The process which the system must perform is called *coordinated query answering*; it is described next. For correctness, it is necessary to ensure that the underlying database is not changed during the answering process.

Grounding the queries: Coordinated query answering makes use of two technical concepts – *valuations* and *groundings*. If q is a query in the intermediate representation and the current database is D, a valuation is simply an assignment of a value from D to each variable of q. For example, on the database in Figure 3.2 (a), Ron's query has three valuations: x can be mapped to either 122, 123 or 134. Every valuation of a query is associated with a *grounding*, which is q itself with the variables replaced by constants following the valuation. We use the terms "grounding" and "grounded query" interchangeably.

Let Q be the set of queries to be evaluated in a coordinated manner. In the description that follows, we make use of $\mathcal{G}$, the set of groundings of the queries on the database. It is important to understand that evaluation does not require that $\mathcal{G}$ be materialized; indeed, our evaluation algorithm presented in Section 3.4 does not materialize it. However, for the purpose of explaining the semantics, $\mathcal{G}$ is a useful tool.

Figure 3.3 (b) shows the set $\mathcal{G}$ obtained by grounding Ron and Harry's queries on the database in Figure 3.2 (a). The bodies of the groundings are no longer needed and can be discarded.

Finding the answers: At a high level, the evaluation is a search for a subset $\mathcal{G}' \subseteq \mathcal{G}$ such that $\mathcal{G}'$ contains at most one grounding of each query and the groundings in $\mathcal{G}'$ can all mutually satisfy each other's postconditions. That is, if all the heads of the groundings in $\mathcal{G}'$ were combined into a set, this set would contain all the postconditions. Any set of groundings satisfying this property is called a *coordinating set*. Once such a $\mathcal{G}'$ is found, the evaluation produces an answer relation which consists of the union of all the head atoms in $\mathcal{G}'$ (the answer may consist of more than one relation – this will happen if the head atoms refer to more than one relation, i.e. the original queries mention more than one `ANSWER` relation).

In the example, the initial set $\mathcal{G}$ is as shown in Figure 3.3 (b). Groundings 1 and 4, as well as groundings 2 and 5, are suitable coordinating subsets $\mathcal{G}'$. Either of them may be used to generate the answer relation and return answers to the respective queries.

It is possible that the selected $\mathcal{G}'$ might not contain any groundings for some queries. This event can be thought of as a statement that those queries could not be answered; it is up to the programmer to determine how to handle this case in the transaction code.

Guarantees on answering: In general, multiple suitable coordinating sets $\mathcal{G}'$ may exist. This raises the question of what requirements one should place on evaluation. It is clearly desirable that some $\mathcal{G}'$ be found unless none exists, and perhaps also that the $\mathcal{G}'$ chosen be maximal, i.e. contain groundings of as many queries as possible. However, as we show next, there are fundamental limitations on the guarantees that we can provide efficiently.

Definition 3.2.1 (CQA). *Let the problem instance consist of a set Q of entangled queries and a database D. The problem is to evaluate Q on D, and to return a nonempty answer if one exists. More formally, the problem is to determine whether there exists a coordinating set $\mathcal{G}' \subseteq \mathcal{G}$, where $\mathcal{G}$ is the set of all groundings for Q on D, containing at most one grounding of each query from Q.*

It turns out CQA is NP-complete. This is unsurprising, as each query in Q has a body that is a conjunctive query, and the combined complexity of evaluating conjunctive queries is NP-complete [20]. The complexity that arises in coordinated query answering, however, is orthogonal to the potential blowup due to the evaluation of the bodies. To demonstrate this, we prove NP-completeness of the following restricted version of the problem where all body queries consist of a single atom and are thus tractable.

Definition 3.2.2 (CQA*). *Let the problem instance consist of a set Q of entangled* *queries, each with a single-atom body, and a database D. The problem is to evaluate Q on D, and to return a nonempty answer if one exists.*

In the case of instances of CQA*, the size of the instance is proportional to the total number of groundings of all queries in Q put together, so no blowup due to grounding occurs. Unfortunately, even in this case, coordinated query answering remains intractable.

Theorem 3.2.3. *CQA* is NP-complete. This holds even if all queries are additionally restricted to have single-atom heads and postconditions.*

Proof. The proof follows a reduction from a custom NP-complete graph-theoretic problem which we call the CNRC (Cycle with no Repeated Colors) problem.

Definition 3.2.4 (Vertex-Colored Digraph)**.** *A vertex-colored digraph is a 3-tuple (V, E, C), where V is a set of vertices $\{v_1, v_2, \ldots, v_n\}$, E is a set of directed edges $\{(v_{a_1}, v_{b_1}), \ldots, (v_{a_m}, v_{b_m})\}$, and $C : V \rightarrow \{1, 2, \ldots, c\}$ is a coloring on vertices.*

Definition 3.2.5 (CNRC)**.** *Given a colored digraph (V, E, C), the Cycle with No Repeated Colors Problem (CNRC) is that of determining whether (V, E, C) contains a directed cycle $v_1, v_2, \ldots, v_l$ s.t. if $1 \leq i < j \leq l$ then $C(v_j) \neq C(v_l)$.*

Theorem 3.2.6. *CNRC is NP-complete.*

The proof for NP-completeness of CNRC is given in the Appendix.

The basic idea behind the proof of Theorem 3.2.3 is to take an instance of CNRC and associate one entangled query with each possible color. Each edge in the graph will be represented by a possible grounding of a entangled query.

Formally, let (V, E, C) be a vertex-colored digraph. We define a table `Edges` such that for every edge $(v_a, v_b) \in E$ there is an entry $(a, b, C(v_a))$ in `Edges`. For each possible color $i = 1, 2, \ldots, c$ we define a new entangled query

$$\{\texttt{Cycle}(b)\}\ \texttt{Cycle}(a)\ \text{:-}\ \texttt{Edges}(a, b, i)$$

We claim that a nonempty coordinating set $\mathcal{G}$ containing at most one grounding of each query in $\mathcal{Q}$ exists if and only if the graph (V, E) contains a cycle with no repeated colors. First suppose that there exists a nonempty coordinating set $\mathcal{G}$. Then the coordinating set must contain a grounding whose body is the atom $\texttt{Edges}(x_1, x_2, C(v_{x_1}))$. Since this grounding's postcondition is satisfied, $\mathcal{G}$ must also contain a grounding that has the body atom $\texttt{Edges}(x_2, x_3, C(v_{x_2}))$. This process can be continued ad infinitum to obtain a sequence of (non-unique) vertex indices $x_1, x_2, \ldots$ such that $(v_{x_i}, v_{x_{i+1}}) \in E$ for all integers $i \geq 1$. By the infinite pigeon-hole principle, there must be some vertex that appears at least twice. Hence, there must be indices s, t with $s \leq t$ such that $x_s, x_{s+1}, \ldots, x_t$ are all distinct and $x_{t+1} = x_s$. We claim that the vertices $x_s, x_{s+1}, \ldots, x_t, x_{t+1}$ form a cycle with no repeating colors. By construction, $(x_i, x_{i+1}) \in E$ for each $i \in [s, t]$. Furthermore, the indices $x_s, x_{s+1}, x_{s+2}, \ldots, x_t$ are all distinct, which means that they must have originated from the heads of different queries. Since each query is associated with its own unique color, it follows that $x_s, x_{s+1}, \ldots, x_t, x_{t+1}$ must be a cycle with no repeating colors, as promised.

Now suppose that there exists a cycle with no repeating colors, say with vertex indices $x_1, x_2, \ldots, x_n, x_{n+1} = x_n$. We claim that there must exist a nonempty coordinating set. Consider the set $\mathcal{G}$ which contains precisely those groundings whose bodies contain the atoms $\texttt{Edges}(x_i, x_{i+1}, C(v_{x_i}))$ for $i = 1, 2, \ldots, n$. By assumption, every such edge really is present in the table. Furthermore, each edge must originate from the body of a different user's entangled query because the colors $C(x_1), C(x_2), \ldots, C(x_n)$ are distinct. This completes the proof.

3.3 Query answering in practice

The main reason for the complexity of entangled query evaluation indicated in Theorem 3.2.3 is not actually the choice of the data values – such as specific flight numbers or hotel rooms. The complexity is due to the fact that if we consider arbitrary sets of queries, a backtracking search [15, 62] is required to discover the coordination *structure*, that is, the way the queries (and their respective groundings) match up together. Moreover, sometimes this coordination structure is not unique. The need to search for the coordination structure can be better understood using the graph encoding from the proof of Theorem 3.2.3; in this setting, it corresponds to a search for the "template" of the desired cycle, i.e. the length and specific sequence of colors involved.

Fortunately, real-world users are very unlikely to generate sets of entangled queries that encode complex constraint satisfaction. In fact, the sets of queries that they do generate are likely to have a very specific structure. It turns out that we can put this knowledge to good use in developing an efficient evaluation algorithm. In this section, we formalize this additional structure and explain why it allows tractable evaluation with respect to the data complexity.

3.3.1 Safe and Unique coordination

We argue that in most practical scenarios, the coordination structure that users express through entangled queries has two formal properties: it is *safe* and *unique*. We informally introduce each of these properties in turn before formalizing them and explaining how they jointly guarantee tractability of evaluation.

We begin with the notion of *safe* coordination. Consider Ron and Harry's example queries from our running example. Each query has a clear coordination

$$\begin{aligned}
\{\texttt{R}(\texttt{Harry}, x)\}\ \texttt{R}(\texttt{Ron}, x) &\ \text{:-}\ \texttt{F}(x, \texttt{Paris})\\
\{\texttt{R}(\texttt{Harry}, y)\}\ \texttt{R}(\texttt{Hermione}, y) &\ \text{:-}\ \texttt{F}(y, \texttt{Athens})\\
\{\texttt{R}(f, z)\}\ \texttt{R}(\texttt{Harry}, z) &\ \text{:-}\ \texttt{F}(z, w) \wedge \texttt{Friend}(\texttt{Harry}, f)
\end{aligned}$$

(a)

$$\begin{aligned}
\{\texttt{R}(\texttt{Harry}, x)\}\ \texttt{R}(\texttt{Ron}, x) &\ \text{:-}\ \texttt{F}(x, \texttt{Paris})\\
\{\texttt{R}(\texttt{Ron}, y)\}\ \texttt{R}(\texttt{Harry}, y) &\ \text{:-}\ \texttt{F}(y, \texttt{Paris})\\
\{\texttt{R}(\texttt{Harry}, z)\}\ \texttt{R}(\texttt{Frank}, z) &\ \text{:-}\ \texttt{F}(z, \texttt{Paris}) \wedge \texttt{A}(y, \texttt{United})
\end{aligned}$$

(b)

Figure 3.4: (a) An unsafe set of queries (b) A set of queries which is not unique

partner. This means there is one clear desired global outcome: both Harry and Ron receive the details of a United flight to Paris. Suppose, however, that we extend the database in our flight booking scenario with a `Friend` relation, and that three users – Ron, Harry and Hermione – are mutual friends. Consider the three queries in Figure 3.4 (a). The queries represent the fact that Ron wants to coordinate with Harry on a flight to Paris, Hermione wants to coordinate with Harry on a flight to Athens, and Harry is happy to coordinate with any friend on any flight.

This set of queries does not fully specify the structure of the desired coordination. Harry's query has two potential queries in the set that could be its coordination partners; however, his query requires a single tuple as an answer. There are two possible coordination outcomes that satisfy *some* users: either Harry flies with Ron or he flies with Hermione. However, there is no outcome that satisfies all users, and it is unclear how the system might choose between the two outcomes above.

To understand what it means for a coordination structure to be *unique*, consider the three queries shown in Figure 3.4 (b). Here Harry and Ron wish to coordinate on a flight to Paris as before. In addition, Frank wishes to coordinate with Harry on a flight to Paris, but only if the airline is United. Depending on the flight database, there are several possibilities for coordination here. First, it may be possible to book all three users on a United flight. Of course, it is possible that no suitable United flights exist. In this case, Harry and Ron may still be able to coordinate and fly with another airline. The coordination structure here is safe – each query has a unique coordination partner – but it is not unique. There are proper subsets of the entire set of queries that may be able to coordinate "locally" even if the entire set cannot.

We next formalize the two above notions.

Safety

Formally, a safe set of queries can be characterized in terms of logical unifiability between various head and postcondition atoms of the queries in the set. Consider two relational atoms containing constants and variables that involve the same relation. They are unifiable unless they contain different constants for the same attribute value; for example, $\mathtt{R}(x, y)$ and $\mathtt{R}(z, z)$ are unifiable whereas $\mathtt{R}(2, y)$ and $\mathtt{R}(3, z)$ are not. We call a set of queries $\mathcal{Q}$ *unsafe* if it contains a query q with a postcondition atom that is unifiable with two (or more) head atoms found in $\mathcal{Q}$. These can be either head atoms of two different queries, or two head atoms of the same query. Evaluation of such queries is intractable and leads to degradation in the performance of the system.

For example, in Figure 3.4 (a), Harry's query has a postcondition atom $\mathtt{R}$(f,z) which unifies with the head of Ron's query as well as the head of Hermione's query. Therefore, the set of queries is unsafe.

If presented with a set of queries which is unsafe, the system has several options. Ideally, the problem would be pointed out to the users involved and they would receive feedback allowing them to reformulate their queries. Alternately, the system could remove queries from the set until the remaining set was safe. A simple way to do this is to iterate over the query set and search for queries q with postconditions that unify with more than one head atom. All such queries q would be removed from the set when found. This procedure is not in general Church-Rosser, but it is simple and can be performed efficiently. More sophisticated strategies for query removal may be appropriate in particular application settings.

Uniqueness of the coordination structure

The formal definition of safety involves excluding queries whose postconditions unify with more than one head. Uniqueness of the coordination structure, on the other hand, has to do with heads that unify with more than one postcondition, as seen in the three queries in Figure 3.4 (b): the head atom of the second query, $\texttt{R}(\texttt{Harry}, y)$ unifies with the postcondition atoms of both the first and third query. However, the restriction required for uniqueness of coordination structure (UCS) is not as straightforward as excluding all queries with such heads; sometimes these types of configurations can be permitted. Intuitively, the problem is due to the fact that a subset of the queries can coordinate separately of the rest.

To define the UCS property for a set of queries, we use a simplified version of the unifiability graph that will be introduced in more detail in Section 3.4. Construct a graph with a node for every query in the system. Draw an edge from node q_i to q_j if a head atom of q_i unifies with a postcondition atom of q_j. Intuitively, if there is a path from query q_k to q_l, this means that groundings of query q_l require groundings of q_k for satisfaction, directly or transitively.

We can use this graph to define UCS. We say that a set of queries has the UCS property if every node in its simplified unifiability graph belongs to a strongly connected component of the same graph. This excludes the type of behavior shown in Figure 3.4 (b). The simplified unifiability graph for this set of queries has three nodes, one for each query. There are three edges – edges in both directions between Harry and Ron's queries, and an additional edge from Harry's query to Frank's query. Thus, Frank's query does not belong to a strongly connected component of the graph.

An interesting property is that a set of queries could satisfy the UCS property even though a query in the set is unsafe. For example, the third query shown in Figure 3.4 (a) is part of the strongly connected component of the graph although it is unsafe.

3.3.2 Tractable evaluation

In settings where the coordination structure is both safe and unique, efficient evaluation is possible.

Theorem 3.3.1. *If a set of entangled queries* $\mathbf{Q}$ *is safe and UCS, then all the queries can be evaluated in PTIME with respect to data complexity.*

In Section 3.4, we prove Theorem 3.3.1 by outlining an algorithm to perform query evaluation in PTIME. The intuition for why efficient evaluation is possible is that the coordination structure can be discovered efficiently. If we construct a graph based on the unifiability of the head and postcondition atoms of the query, the strongly connected components of the unifiability graph correspond to sets of queries that are coordination partners and require each other's postconditions during evaluation.

Within each such group, the specific way in which the queries match is unique. It is therefore possible to collect the queries together into a big query that specifies a single joint outcome based on the way they match. This is explained in much greater detail in Section 3.4, but as an example, Harry and Ron's queries from the introduction can be combined into this postcondition-free query:

$$\texttt{R}(\texttt{Ron}, x) \wedge \texttt{R}(\texttt{Harry}, x) :- \ \texttt{F}(x, \texttt{Paris}) \wedge \texttt{A}(x, \texttt{United})$$

This query specifies that the system should find a United flight to Paris and return the two answer tuples to Harry and Ron.

In the evaluation process as outlined above, safety guarantees tractability, by ensuring that there is a unique way to combine the queries in each strongly connected component into a bigger query. The UCS property guarantees correctness: we know that we will not miss any possible answers (i.e. coordinating sets of groundings) that involve proper subsets of a set of matching queries, as explained in our discussion of the queries in Figure 3.4 (b).

3.4 The evaluation algorithm

We now introduce our algorithm for coordinated query answering. Within our system, this algorithm is implemented in the coordination module as explained in Section 3.5.1. It is invoked by the coordination middleware, either automatically at regular intervals or through explicit requests. Upon invocation, the algorithm operates on a snapshot of the database and on a fixed set Q of queries. The set Q is assumed to be safe; if necessary, a simple check can be run on Q to ensure safety.

The algorithm has two main phases: query matching and evaluation proper. Query matching discovers the coordination structure implicit in the individual entangled queries and uses this structure to construct a set of combined queries. Once each combined query is available, it is sent to the database for evaluation; each answer to this query corresponds to a set of answers to the individual entangled queries. The first (or any other) combined query answer can be used to produce the individual answers.

3.4.1 Query Matching

Query matching discovers the coordination structure implicit in the set of entangled queries. In most cases, as discussed, users submit small groups of queries that match only each other. That is, the structure consists of a potentially large number of small, disconnected groups of queries that will coordinate only internally.

The query matching phase discovers this structure in two steps. First, it identifies the disconnected, independent groups of queries. In doing so, it partitions Q into a set of components which can subsequently be processed independently and in parallel. We call this phase the partitioning phase and describe it in Section 3.4.1.

Next, the algorithm works on each group of queries to discover the actual coordination by determining how the query heads and postconditions match. We refer to this phase as matching (proper) and describe it in Section 3.4.1.

All stages of this process make use of a data structure called the *unifiability graph* that represents certain dependencies among the queries in Q with respect to matching. We begin by introducing this graph and explaining how it is constructed. We then discuss how the subsequent phases make use of it.

The unifiability graph

The *unifiability graph* of a set of queries Q is a multi-digraph (directed multigraph) that contains a distinct node $N(q_i)$ for each query q_i in Q. There is an edge from query node $N(q_i)$ to query node $N(q_j)$ for each pair of atoms (h, p) such that h is a head atom of q_i, p is a postcondition atom of q_j, and h unifies with p. For the remainder of this section, we use q_i to represent both a query in Q and the corresponding node in the unifiability graph.

For every query q_i in Q, let $\texttt{INDEGREE}(q_i)$ denote the indegree of the corresponding graph node, and let $\texttt{PCCOUNT}(q_i)$ equal the number of postconditions of query q_i. Safety guarantees that there will be at most one edge into a graph node q_i for each postcondition of q_i. This means that for every query q_i in Q,

$$\texttt{INDEGREE}(q_i) \leq \texttt{PCCOUNT}(q_i)$$

Equality holds if and only if every postcondition atom of q_i unifies with a head atom of some query.

For instance, suppose Q consists of the three following queries:

$$q_1 : \ \{\texttt{R}(x_1) \wedge \texttt{S}(x_2)\} \ \texttt{T}(x_3) :\!- \texttt{D1}(x_1, x_2, x_3)$$

$$q_2 : \ \{\texttt{T}(1)\} \ \texttt{R}(y_1) :\!- \texttt{D2}(y_1)$$

$$q_3 : \ \{\texttt{T}(z_1)\} \ \texttt{S}(z_2) :\!- \texttt{D3}(z_1, z_2)$$

Then the unifiability graph is as shown in Figure 3.5 (a). We will use this set of three queries as our running example for this section.

Partitioning

The unifiability graph allows Q to be partitioned into subsets that can be processed separately and in parallel. These partitions are precisely the connected components of the unifiability graph; for convenience, we refer to the queries corresponding to a connected component of the unifiability graph as a *component* of Q. Suppose that queries q_1 and q_2 are in different components of Q. Then any coordinating set that contains groundings of both q_1 and q_2 can be broken into two smaller disjoint coordinating sets, one of which contains q_1 and the other of which contains q_2. All subsequent stages of evaluation can therefore be performed separately on each component of Q. Partitioning the graph has other potential benefits in addition to the performance advantages associated with increased parallelization and smaller search spaces. For instance, it has security benefits. By analyzing the unifiability graph, an implementation of our system could provide guarantees about the interaction between different queries in the system. A system sensitive to privacy could partition the workload by grouping queries into sets of "trusted and sensitive," "trusted but not sensitive," or "untrusted" queries and ensure that no component of Q could contain both a "trusted and sensitive" and an "untrusted" query.

Unifier Propagation

At the core of our algorithm is an iterative process that identifies and removes unanswerable queries, i.e. those that have no chance to participate in a coordinating set. Fundamental to the algorithm is the observation that a query with a postcondition that does not unify with any query's head cannot have a grounding that participates in a coordinating set. Any such query can therefore be safely disregarded. We can identify such queries using our unifiability graph: a query node $N(q_i)$ can be safely removed from the graph if its indegree is strictly less than the number of postconditions of q_i.

Unifier propagation requires that no variable name can appear in more than one query. If the initial Q does not satisfy this property, it is easy to enforce it by renaming variables as needed. For the remainder of this section, we assume that each variable is indeed unique to a single query. Let `Val` denote the set of all constants and variables occurring in Q.

Unifiers The matching algorithm associates a *unifier* $U(n)$ with each node n in the unifiability graph. A unifier is a constraint on the valuations of the variables in `Val`. Formally, it is a partition of a subset of `Val` which contains at most one constant per partition class. It can be represented as a set of subsets of `Val`. For example, $\{\{x, 3\}, \{y, z\}\}$ is a unifier specifying that in any permitted valuation, the variable x must have value 3 and the variables y and z must have the same value.

Given unifiers u_1 and u_2, the *Most General Unifier* of u_1 and u_2, denoted $mgu(u_1, u_2)$, is the most general (least restrictive) unifier that enforces all the constraints imposed by each u_i. In general, $mgu(u_1, u_2)$ may not exist, but if it does exist then it is unique. For instance, there is no most general unifier for the unifiers $\{\{x, 3\}\}$ and $\{\{x, 4\}\}$; if one existed, it would need to restrict valuations so that x was equal to both 3 and 4.

Given two unifiers u_1 and u_2, it is possible to compute $mgu(u_1, u_2)$ – or determine that it does not exist – using standard methods. An optimized implementation of the MGU procedure based on disjoint-set forests provides strong performance guarantees. If unifiers u_1 and u_2 jointly contain k distinct variables then it possible to compute their most general unifier in expected $O(k \cdot \alpha(k))$ time, where α is the inverse of the Ackermann function.

Cascading effects of unifier propagation If a query node q_i is removed from the graph then we can also remove any node q_j such that a postcondition atom of q_j unifies with a head atom of q_i. This is true because of our safety condition:

we know that each postcondition atom unifies with at most one head atom.

In practice, this means that if a node q_i is removed from the unifiability graph then every successor q_j of q_i may be removed as well. Repeating this argument, we may remove every successor of a successor of q_i, and so on until we have removed all descendants of q_i from the graph. This can be accomplished using a standard graph traversal algorithm such as Breadth-First Search. We assume that there is a function `CLEANUP`(n) that removes an input node and all its descendants from the dependency graph, as well as all edges into and out of those nodes. We also assume that `CLEANUP` removes all of these nodes from the *updates* queue, a data structure whose purpose will be described shortly.

Matching

We are now ready to explain the query matching algorithm proper.

We begin by constructing a unifiability graph for the set of queries Q. For each query q_i in Q, we create a node, and we define a set $U(q_i)$, called the *unifier*, for this node. Intuitively, $U(q_i)$ represents the minimal (least restrictive) currently known constraints on valuations that must hold for any coordinating set that contains a grounding of q_i.

We initialize the unifier $U(q_i)$ of each node q_i to the empty set. For each head atom h of each query q_i we check whether there is a postcondition atom p of a query q_j that unifies with it. If such a p exists then we create an edge from q_i to q_j in the unifiability graph. We also update $U(q_j)$ to be the MGU of $U(q_j)$ and the most general unifier of p and h. If no such h exists or no MGU exists then the query q_i is unsatisfiable, and we may run `CLEANUP` to remove it and all its descendants from the graph.

The unifiability graph can be generated in a straightforward but inefficient manner by trying to unify each postcondition with each head in our entire input set of queries. This process can be made more efficient by building indices, but doing so is non-trivial. For example, consider the atoms `Reserve (Ron, x)` and `Reserve (Harry, y)`. Clearly, a unifier does not exist for these atoms despite the fact that they point to the same relation. Interestingly, we can *attempt* to reduce the number of these matchings by simply replacing the variables in every atom by a unique constant Δ. We then build an index on all heads in Q of the following form:

```
(Relation, Parameter, Value) → [List of Atoms]
```

A lookup for a postcondition atom `Reserve (Harry, y)` involves a seek on the index for `(Reserve, 1, Harry)` and `(Reserve, 1, Δ)`. Formally, if $\mathcal{L}$ denotes the lookup function on the index and $\mathcal{A}$ represents the set of atoms, an atom $R(v_1 \ldots v_n)$ can only unify with

$$\mathcal{A} \cap \bigcap_{constants\ v_i} (\mathcal{L}(R, i, v_i) \cup \mathcal{L}(R, i, \Delta))$$

Such an index structure does not provide us with any guarantee on complexity. Indeed, we expect it to perform poorly when queries have many variables. However, a query set with a very large number of variables is highly likely to be unsafe: postconditions and heads that contain mostly variables rather than constants will typically unify with each other densely. In practice, therefore, this type of index is immensely useful.

In building the graph, we iteratively removed any query containing a postcondition that did not unify with some head atom. This fact, together with our assumption that Q is safe, is sufficient to guarantee that that for each postcondition p of each query q_i in the graph there is exactly one other query q_j with a head h that unifies with p. This establishes a local satisfaction of constraints

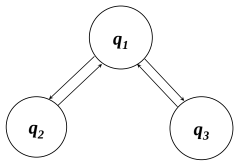
Algorithm 8: Matching on a unifiability graph G

```
1:  updates := queue containing all nodes in G
2:  while updates is not empty do
3:    parent := DEQUEUE(updates)
4:    for child in successors of parent do
5:      U(child) := MGU(U(parent), U(child))
6:      if U(child) was changed then
7:        if U(child) = NIL then
8:          CLEANUP(child)
9:        else
10:          ENQUEUE(updates, child)
```

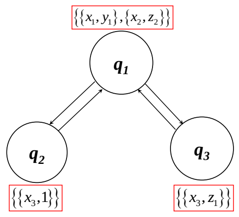
for each of the remaining nodes in the dependency graph. The algorithm next propagates these constraints using the structure of the unifiability graph. More specifically, if a postcondition of query q_j requires the head of some query q_i for satisfaction, the coordinating set cannot contain a grounding of q_j unless both q_j's existing constraints and q_i's constraints hold.

Unifier propagation is an iterative procedure that runs on each component of the unifiability graph. As it runs, it performs two tasks. First, it discovers the coordination structure, i.e. how the queries match with respect to satisfying each other's postconditions. As it does this, it updates the unifiers associated with the graph nodes to reflect the current known constraints on valuations that are required for this query to be answerable. Simultaneously, the algorithm discovers and removes unanswerable queries from the graph.

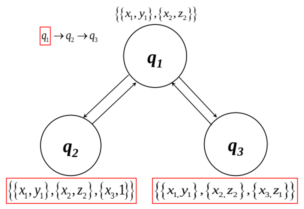
The propagation procedure is shown in Algorithm 8. At a high level, it pushes unifier information forward along edges. If a unifier does not exist for some node q_i then the `CLEANUP` function is invoked on q_i, removing it and all its descendants from the unifiability graph and the *updates* queue. The intuition is that such a node corresponds to an unanswerable query, and any descendants of this node represent queries that relied on a postcondition of q_i for satisfac-



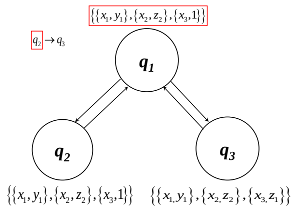
(a) Basic unifiability graph



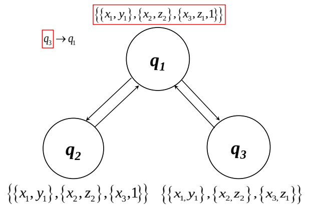
(b) Computing initial unifiers


$\{\{x_1, y_1\}, \{x_2, z_2\}\}$
$q_1 \to q_2 \to q_3$
q_1
q_2
q_3
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1\}\}$

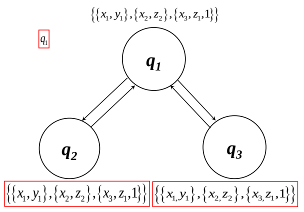
(c) Processing q_1


$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, 1\}\}$
$q_2 \to q_3$
q_1
q_2
q_3
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1\}\}$

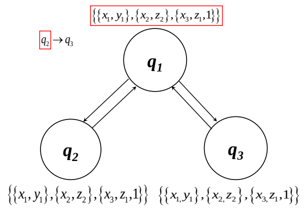
(d) Processing q_2


$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
$q_3 \to q_1$
q_1
q_2
q_3
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1\}\}$

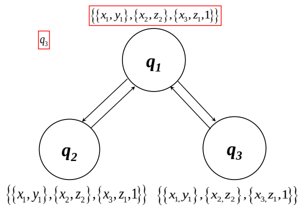
(e) Processing q_3


$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
q_1
q_1
q_2
q_3
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$

(f) Reprocessing q_1


$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
$q_2 \to q_3$
q_1
q_2
q_3
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$

(g) Reprocessing q_2


$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
q_3
q_1
q_2
q_3
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$
$\{\{x_1, y_1\}, \{x_2, z_2\}, \{x_3, z_1, 1\}\}$

(h) Reprocessing q_3

Figure 3.5: A sample run of matching

tion, so are also unanswerable. Whenever the unifier of a node is updated, that node is added to the *updates* queue so that the change can be propagated to the node's children. This propagation of unifier information continues until no new information is propagated by any of the nodes and the *updates* queue becomes empty.

The execution of the algorithm on our running example is shown in Figure 3.5. In Figure 3.5 (b), unifiers are computed for all nodes in the graph, and all nodes in the graph are added to the *updates* queue. In Figure 3.5 (c), the first node, q_1, is removed from the head of the queue and information about its constraints is propagated to its successors q_2 and q_3. In 3.5 (d), q_2 is removed from the queue and information about its constraints is propagated to its child q_1. Since q_1 is not currently in the queue, it is added at this point. In 3.5 (e), q_3 is removed from the queue and its constraints are propagated to its child q_1. In 3.5 (f), q_1 is processed again with its new unifier, and information about the update is propagated to q_2 and q_3. In 3.5 (g) and 3.5 (h), the update is propagated to q_1, but since $U(q_1)$ is not changed by the operation, it is not added to the queue.

We now consider a variant of this example in which q_3 has the postcondition $\mathtt{T}(2)$ rather than $\mathtt{T}(z_1)$. In this case, no choice of head atoms for q_1 can simultaneously satisfy the postconditions of q_2 and q_3, so we expect that the matching algorithm should fail. Indeed, immediately before Figure 3.5 (e), $U(q_2)$ will contain the set $\{x_3, 1\}$ and $U(q_3)$ will contain the set $\{x_3, 2\}$. The unifier of q_1 will be updated first to $mgu(U(q_1), U(q_2))$ and then to the unifier of that value with $mgu(U(q_1), U(q_2))$. The last unification will require x_3 to be equal to 1 and 2 simultaneously, and that unification will therefore fail. As expected, the matching algorithm will consequently eliminate the node q_1 and its children q_2 and q_3.

Complexity Analysis

Graph Construction We first analyze the complexity of constructing the unifiability graph. Let H denote the total number of head atoms in all queries in Q, let P denote the total number of postcondition atoms, and let κ denote the greatest number of columns that appears in any single atom in Q. In the absence of any indices, for each head atom h and postcondition atom p in Q, we must check whether h unifies with p; each such check takes expected $O(\kappa\ \alpha(\kappa))$ time. If h is fixed then we must perform this check with P different values of p. Since every query in Q contains at least one postcondition atom, the time required to find all postcondition atoms and perform this loop is expected $O(P\ \kappa\ \alpha(\kappa))$. We must perform this inner loop for H different values of h. Since each query in Q has at least one head atom, finding all the head atoms in the input and iterating over all of them takes expected $O(P\ H\ \kappa\ \alpha(\kappa))$ time.

Unifier Propagation We now analyze the complexity of Algorithm 8. The input is a connected component of the unifiability graph containing nodes $Q' \subset Q$ such that each variable appears in at most one query in Q', as well as a unifier for each node in the graph. Suppose that all queries in the input jointly contain k free variables, and let w be the maximum number of postconditions of any query in Q. Let P be the total number of postcondition atoms in every query in the graph, and n the number of queries in Q'.

We add a node to the *updates* queue only at the very beginning of the algorithm or when its unifier is updated by a call to the MGU function on line 5 of the algorithm pseudocode. First suppose that $k = 0$, i.e. there are no variables in the input. In this case, unification is trivial, unifiers are never changed, and the whole algorithm runs in time proportional to the number of edges in the graph; this is bounded above by $O(P)$ time.

Now suppose that $k > 0$. If a unifier is updated by a call to the MGU function then either the new unifier must contain a constant that the old unifier did not contain or else two sets in the old unifier must be merged together and the total number of sets in the unifier must decrease. This means that if all queries in the input jointly contain k free variables then for each node *child* in Q', the check on line 6 can succeed at most $O(k)$ times. If every node q in the input has indegree at most w then each node can be added to the *updates* queue at most $O(kw)$ times.

It follows that lines 5-12 can be executed at most once $O(kw^2)$ for each node in the graph. Each execution takes expected $O(k \cdot \alpha(k))$ time if we ignore the time spent in the `CLEANUP` function on line 8, so the running time of the loop is expected $O(k^2\ w^2 \cdot \alpha(k))$. The total time spent in the `CLEANUP` function across all calls is at worst linear in the number of nodes in the input. It follows that the entire procedure runs in $O(k^2\ w^2\ \alpha(k) + n + P)$ time. Since every query in the input contains at least one postcondition, this can be simplified to expected $O(k^2\ w^2\ \alpha(k) + P)$ time.

Discussion

We note that at any given time, the unifier of a query node q represents the weakest constraints on variables that must hold in order for there to be a coordinating set of groundings for a subset $Q' \subset Q$ that contains exactly one grounding for each query in Q'. A node is removed from the graph only when this is known to be impossible, either because some of its postconditions can't be satisfied at all or because some subset of its postconditions can't be mutually satisfied by any variable assignment.

This is the best we can do without any knowledge of the records in the database: the unifier of any query that remains in the system after the matching algorithm halts can be satisfied for some valuation of the variables it contains. This means that for each remaining query q there exists a database D, a set of queries $\overline{Q} \subset Q'$, and a coordinating set of groundings $\mathcal{G}$, such that $q \in \overline{Q}$ and $\mathcal{G}$ consists of exactly one grounding for each $q' \in \overline{Q}$.

3.4.2 Constructing and evaluating the combined query

After the matching procedure finishes, we are left with a set of answerable queries $Q = \{q_i\}_{i \in I}$, each associated with a unifier $U(q_i)$, such that Q is a subset of the current component Q' of Q. We compute a global unifier U for the whole set of queries as $mgu(\{U(q_i)\})$. If such a U cannot be computed, evaluation fails for Q' and all the queries in Q' are rejected. If U does exist then it can be expressed as a conjunction of equality statements relating the variables and constants involved; call this conjunction φ_U.

At this point, the evaluation algorithm creates a combined query using Q and φ_U. Let B_i denote the body of query q_i, and let H_i denote the conjunction of its head atoms. Then the combined query q^* is

$$\bigwedge_i H_i \coloneq \bigwedge_i B_i \wedge \varphi_U$$

That is, the body of q^* is the conjunction of all the bodies of the original queries, together with equality atoms that encode the constraints in u. The head of q^* is the conjunction of the original query heads.

In our running example illustrated in Figure 3.5, all query nodes end up with the same unifier after matching. This is

$$\{\{x_1, y_1\},\ \{x_2, z_2\},\ \{x_3, z_1, 1\}\}$$

The required most general unifier U is consequently also

$$\{\{x_1, y_1\},\ \{x_2, z_2\},\ \{x_3, z_1, 1\}\}$$

A suitable corresponding φ_U is

$$x_1 = y_1 \wedge x_2 = z_2 \wedge x_3 = z_1 \wedge x_3 = 1$$

The combined query generated by the system is as follows:

$$\texttt{T}(x_3) \wedge \texttt{R}(y_1) \wedge \texttt{S}(z_2) \coloneq \texttt{D1}(x_1, x_2, x_3) \wedge \texttt{D2}(y_1) \wedge \texttt{D3}(z_1, z_2)$$
$$\wedge x_1 = y_1 \wedge x_2 = z_2 \wedge x_3 = z_1 \wedge x_3 = 1$$

As this example makes clear, q^* can be simplified making use of the information in φ_U. Our example query is equivalent to the following query:

$$\texttt{T}(1) \wedge \texttt{R}(x_1) \wedge \texttt{S}(x_2) \coloneq \texttt{D1}(x_1, x_2, x_3) \wedge \texttt{D2}(x_1) \wedge \texttt{D3}(1, x_2)$$

Once q^* is constructed, it can be sent to the database for evaluation. Each answer to q^* is a valuation of the variables in q^* that corresponds to a set of fully grounded head atoms. Only one such valuation is necessary to answer the entangled queries, so q^* may be equipped with a `LIMIT 1` clause. Once an answer is available, the fully grounded head atoms can be used to generate answers for the individual queries from Q in a straightforward manner.

3.5 Youtopia System

In this section, we describe the system that we built to provide end-to-end support for entangled queries. We also describe a real world travel booking application that we designed and implemented to establish the usability of entangled queries for coordination.

3.5.1 D3C Engine

Designing an entire system to provide end-to-end coordination support is a major research challenge. For instance, all levels of the system must handle not just coordination success, but coordination failure as well. Suppose Ron submits his query as in our first example, but Harry's matching query never arrives; the system needs a suitable mechanism for dealing with this, ultimately sending a message to the transaction code that the query is not answerable. As another example, suppose Ron and Harry do coordinate, but Ron's transaction aborts before he makes the booking. The coordination has created a dependency between their transactions which must be considered during recovery.

In [34], the authors argue that designing for D3C raises questions about the very foundations of database system design. Coordination, by definition, requires communication between user programs. As such, it is a breach of isolation, which is a cornerstone of the transaction abstraction. If transactions are no longer isolated, this has fundamental implications for the overall system architecture at all levels. A model that integrates entangled queries into transactions is described in [23].

Figure 3.6 gives the outline of the portion of our system which is directly involved in handling entangled queries. The design is closely tied to the life cycle of the entangled queries, from the moment the query is generated until the answers are returned.

Entangled queries can, in principle, be input by hand, but normally they are generated by a front end web interface, just like regular (non-entangled) queries. Once a query is generated, it is passed to a suitable layer for answering.

From the perspective of the application, the coordinated answering is an asynchronous process. An individual query may not in general be answerable until other, partner queries are available. The middleware layer provides to the application an asynchronous query answering abstraction with callback functionality. Such an abstraction is needed due to the misalignment between the asynchronous query *submission* by the application code and the synchronous entangled query *answering* by the coordination module.

It is unrealistic for an entangled query to wait an arbitrary amount of time for a coordination partner. To deal with this, the system has a notion of *query staleness*; when a query becomes stale, it is removed from the list of pending queries and its evaluation is considered to have failed. Any further handling of the query is up to the programmer in application code. Staleness can be defined in a variety of ways; a timeout mechanism or manual user intervention are two possibilities.

Below the middleware layer, a dedicated module actually performs the coordination. The structure of this module directly mirrors that of the algorithm presented in Section 3.4. It receives a stream of queries and constructs the unifiability graph using suitable indices over the queries. Subsequently, each component of the graph can be processed by an independent server thread, which performs the actual query matching and generates a combined query. This combined query is then sent to the database for evaluation. The DB query optimizer can apply traditional query optimization techniques in evaluating this combined query. Once the coordination module computes answers to the individual entangled queries, these answers are returned back to the application code.

At present, we have a full implementation of the coordination module. The implementation consists of a server which can accept connections and queries from a hundred clients. The evaluation algorithm can be executed periodically in a set-at-a time fashion (after specific time intervals or after a fixed number of queries). Alternately, it can be executed incrementally upon submission of every query. On the arrival of a new query in the system, the unifiability graph may be updated and only certain partitions may require updates. The incre-

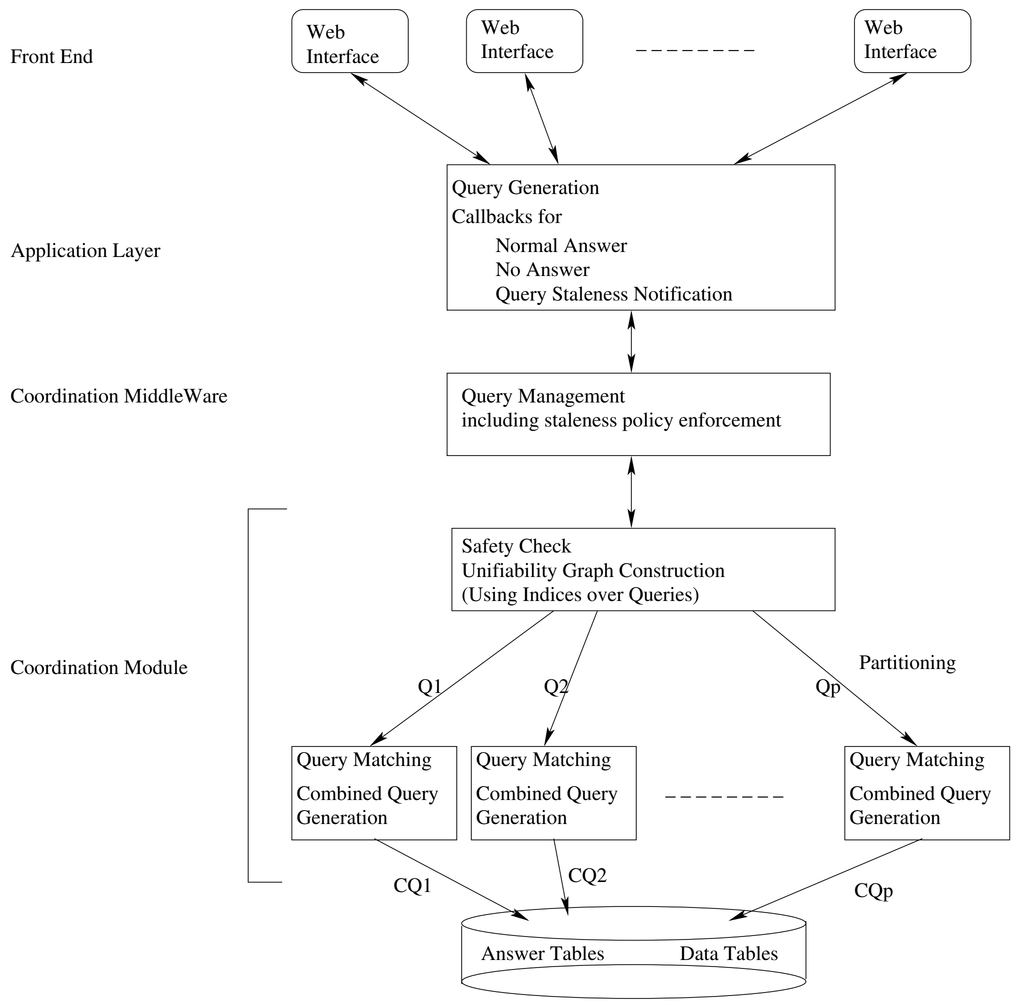


Figure 3.6: D3C Engine based on entangled queries

mental evaluation requires each partition to store the partial matching unifiers and continues the matching algorithm from this state with the addition of a new query. A parameter in our implementation allows us to switch between the two. Section 3.6.2 discusses the impact of using each of these approaches.

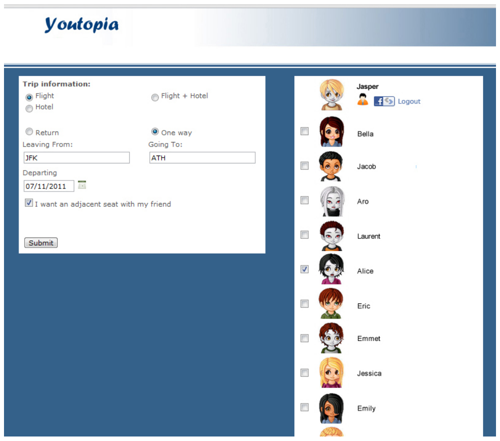
The system is implemented in Java 1.6.0. The implementation uses JDBC to connect to a MySQL database system (version 4.1.20).

3.5.2 Real-World application

We implemented entangled queries in an actual travel Web site that allows users to coordinate travel and hotel reservations with their Facebook friends. The application itself follows a standard three-tier architecture. The graphical frontend runs in a browser; it gives an interface to all the functionality provided by the middle tier. At the middle tier, we have implemented application logic to handle the standard functionality of a travel Web site such as searching for flights and hotels, selecting specific flights and hotels, and to create and coordinate new travel reservations based on the users list of friends that is populated using the Facebook API. The application logic also contains an account view where a user can see pending or confirmed reservations.

Coordinated travel is very simple using our application. In order for Hermione to make joint travel bookings with Harry, the workflow is as follows:

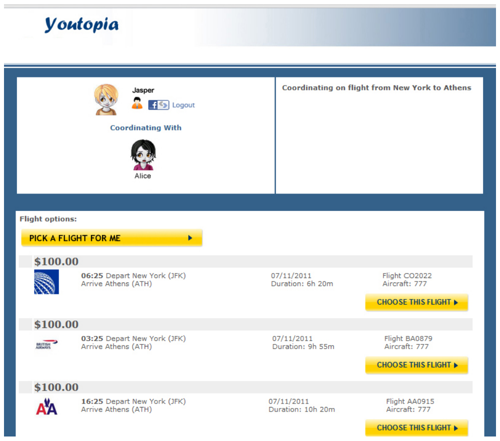
1. Hermione logs into the system and is led to her travel booking page. She can see a form to specify her travel criterion along with a list of Facebook friends on the right panel (Figure 3.7a).
2. From the left panel, she chooses her route and dates of travel. She then choose Harry in the right panel and her travel partner. On clicking submit, she is led to the flight selection page.
3. On the flight selection page (Figure 3.7b), she either chooses a flight of her choice, or asks the system to find a flight that fits her criterion.
4. On submission of her request, Hermione's booking request is added to the system (Figure 3.7c). She is given a request code along with a confirmation that her request has been queued.



Trip information:
[✓ Flight] Flight + Hotel
Hotel

Return [✓ One way]
Leaving From: JFK
Going To: ATH
Departing 07/11/2011
[✓] I want an adjacent seat with my friend

(a)

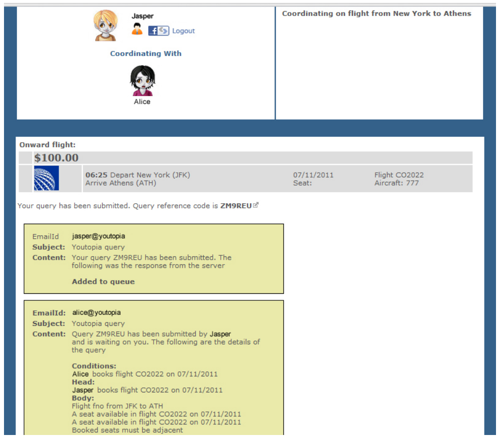


Coordinating on flight from New York to Athens

Flight options:

$100.00			
	06:25 Depart New York (JFK) Arrive Athens (ATH)	07/11/2011 Duration: 6h 20m	Flight CO2022 Aircraft: 777
$100.00			
	03:25 Depart New York (JFK) Arrive Athens (ATH)	07/11/2011 Duration: 9h 55m	Flight BA0879 Aircraft: 777
$100.00			
	16:25 Depart New York (JFK) Arrive Athens (ATH)	07/11/2011 Duration: 10h 20m	Flight AA0915 Aircraft: 777

(b)



Coordinating on flight from New York to Athens

Onward flight:

$100.00			
	06:25 Depart New York (JFK) Arrive Athens (ATH)	07/11/2011 Seat:	Flight CO2022 Aircraft: 777

Your query has been submitted. Query reference code is **ZM9REU**

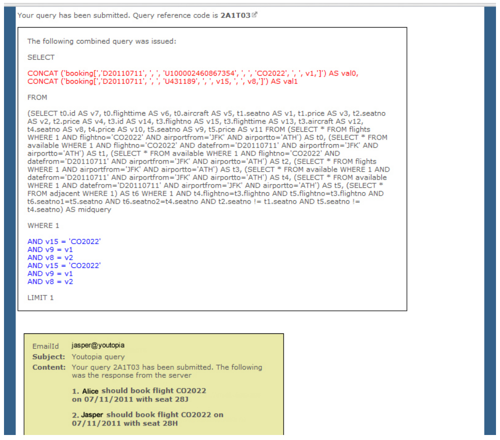
EmailId jasper@youtopia
Subject: Youtopia query
Content: Your query ZM9REU has been submitted. The following was the response from the server

Added to queue

EmailId: alice@youtopia
Subject: Youtopia query
Content: Query ZM9REU has been submitted by Jasper and is waiting on you. The following are the details of the query

Conditions:
Alice books flight CO2022 on 07/11/2011
Head:
Jasper books flight CO2022 on 07/11/2011
Body:
Flight fno from JFK to ATH
A seat available in flight CO2022 on 07/11/2011
A seat available in flight CO2022 on 07/11/2011
Booked seats must be adjacent

(c)


Your query has been submitted. Query reference code is **2A1T03**

```
The following combined query was issued:

SELECT

CONCAT ('booking[','D20110711', ', ', 'U100002460867354', ', ', 'CO2022', ', ', v1,']') AS val0,
CONCAT ('booking[','D20110711', ', ', 'U431189', ', ', v15, ', ', v8,']') AS val1

FROM

(SELECT t0.id AS v7, t0.flighttime AS v6, t0.aircraft AS v5, t1.seatno AS v1, t1.price AS v3, t2.seatno
AS v2, t2.price AS v4, t3.id AS v14, t3.flightno AS v15, t3.flighttime AS v13, t3.aircraft AS v12,
t4.seatno AS v8, t4.price AS v10, t5.seatno AS v9, t5.price AS v11 FROM (SELECT * FROM flights
WHERE 1 AND flightno='CO2022' AND airportfrom='JFK' AND airportto='ATH') AS t0, (SELECT * FROM
available WHERE 1 AND flightno='CO2022' AND datefrom='D20110711' AND airportfrom='JFK' AND
airportto='ATH') AS t1, (SELECT * FROM available WHERE 1 AND flightno='CO2022' AND
datefrom='D20110711' AND airportfrom='JFK' AND airportto='ATH') AS t2, (SELECT * FROM flights
WHERE 1 AND airportfrom='JFK' AND airportto='ATH') AS t3, (SELECT * FROM available WHERE 1 AND
datefrom='D20110711' AND airportfrom='JFK' AND airportto='ATH') AS t4, (SELECT * FROM available
WHERE 1 AND datefrom='D20110711' AND airportfrom='JFK' AND airportto='ATH') AS t5, (SELECT *
FROM adjacent WHERE 1) AS t6 WHERE 1 AND t4.flightno=t3.flightno AND t5.flightno=t3.flightno AND
t6.seatno1=t5.seatno AND t6.seatno2=t4.seatno AND t2.seatno != t1.seatno AND t5.seatno !=
t4.seatno) AS midquery

WHERE 1

AND v15 = 'CO2022'
AND v9 = v1
AND v8 = v2
AND v15 = 'CO2022'
AND v9 = v1
AND v8 = v2

LIMIT 1
```

EmailId jasper@youtopia
Subject: Youtopia query
Content: Your query 2A1T03 has been submitted. The following was the response from the server

1. Alice should book flight CO2022 on 07/11/2011 with seat 28J

2. Jasper should book flight CO2022 on 07/11/2011 with seat 28H

(d)

Figure 3.7: Youtopia Social-Travel Website

5. Harry follows similar steps. If he chose Hermione as a coordination partner and both their constraints are satisfied, the system issues a combined SQL statement for both the users (Figure 3.7d).

6. Harry is given a confirmation along with seats alloted to both Hermione and him. Hermione is also sent an email with this information.

The coordination, from the users perspective, is therefore very simple. The users are not made aware of what goes on in the system. Also, at no point was Hermione asked to email Harry her request id or any other details. The coordination was implicitly determined by their individual preferences and constraints.

This real world application helped us establish the ease of use of entangled queries. It demonstrates the applicability of our approach along with its social utility.

3.6 Experiments

This section outlines our experimental setup and presents results from an experimental evaluation of our implementation of the query evaluation algorithm.

3.6.1 Experimental Setup

To evaluate the system, we use a simulated flight booking scenario in which users want to coordinate their travel plans with their friends. We use the *Slashdot* social network data [41] to establish friendship relationships between users. The graph has 82168 users and 102 airport destinations. We assign a "hometown" airport to each of the users, ensuring as far as possible that that each user has at least half his or her friends living in the same city.

The schema for our system is as follows:

```
Reserve(UserName, Destination)
Friends(UserName1, UserName2)
User(UserName, HomeTown)
```

In the rest of this section, we use `R`, `F` and `U` to denote the Reserve, Friends and User tables respectively. Within this flight booking scenario, we test our system under various different coordination scenarios and with different workloads.

We run all experiments on an Intel Core-2 Duo E8500 3.16GHz processor with 3.2GB of RAM; the reported values are averages over three runs. The standard deviation is less than 2% in each experiment.

3.6.2 Results

We present results from five sets of experiments. The first three are designed to test the scalability of coordination in an increasingly complex set of scenarios; the last two stress-test our query matching and safety check procedures. All experiments use an incremental version of the algorithm unless specified otherwise.

Two-way coordination

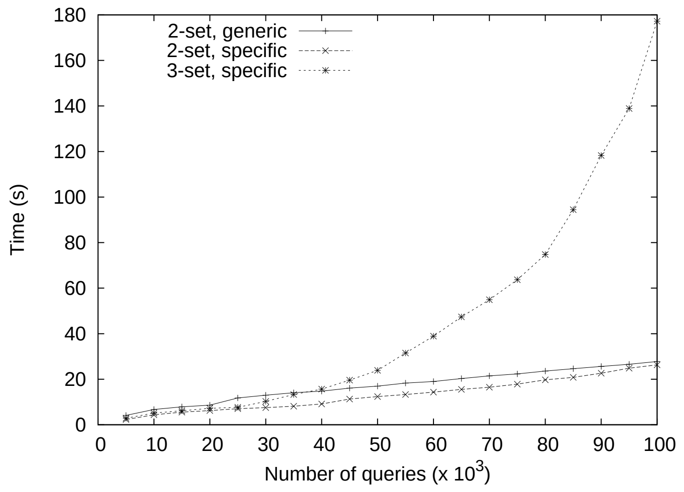


Figure 3.8: Scalability on best-case and random workload

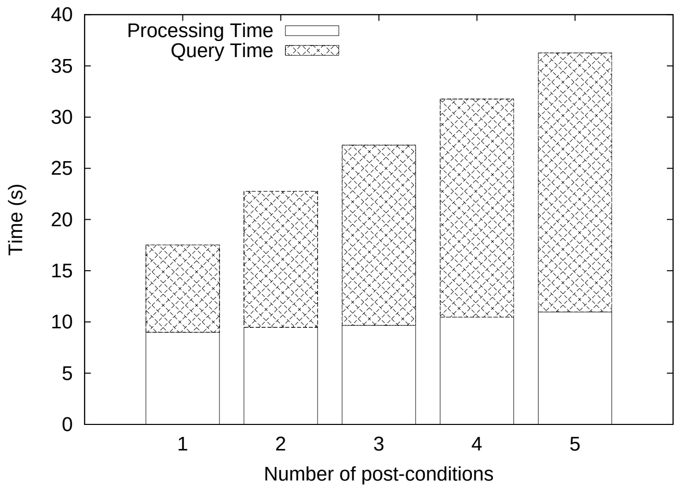


Figure 3.9: Scalability in the number of postconditions

The first experiment tests the scalability of coordinated query answering in a basic scenario where pairs of friends want to coordinate on flights. The query sets used consist of pairs of queries of the following form:

$$\{\mathtt{R}(x,\mathtt{JFK})\}\ \mathtt{R}(\mathtt{Harry},\mathtt{JFK})\ :-$$
$$\mathtt{F}(\mathtt{Harry},x) \wedge \mathtt{U}(\mathtt{Harry},c) \wedge \mathtt{U}(x,c)$$
$$\{\mathtt{R}(x,\mathtt{JFK})\}\ \mathtt{R}(\mathtt{Ron},\mathtt{JFK})\ :-$$
$$\mathtt{F}(\mathtt{Ron},x) \wedge \mathtt{U}(\mathtt{Ron},c) \wedge \mathtt{U}(x,c)$$

The intuition is that the above pair of queries is generated by Harry and Ron who each want to fly to JFK with any of their friends. When generating such query pairs, we ensure that Harry and Ron are friends according to the social network structure, but we do not ensure that they live in the same city. Enforcing only one of these two conditions in query generation allows us to produce queries that have a realistic – not too small and not too large – chance to coordinate.

We vary the size of our query sets from five to one hundred thousand. In addition, to detect any side effects of our incremental query evaluation approach, each run of the experiment is evaluated on a randomly permuted set of mutually coordinating pairs of queries. Figure 3.8 shows our results.

It is interesting to note that although the heads and postconditions of all queries point to the same ANSWER relation, the performance of system is linear in the number of queries. This is due to an artifact of our dataset in which queries coordinate often and the number of "pending" queries in the system does not grow with an increase in the number of queries.

We also test the effect of making the queries more specific. In particular, we eliminate the variables from the postcondition and the head, so that the pairs queries are now of the following form:

$$\{\mathtt{R}(\mathtt{Ron},\mathtt{JFK})\}\ \mathtt{R}(\mathtt{Harry},\mathtt{JFK})\ :-$$
$$\mathtt{F}(\mathtt{Harry},\mathtt{Ron}) \wedge \mathtt{U}(\mathtt{Harry},c) \wedge \mathtt{U}(\mathtt{Ron},c)$$
$$\{\mathtt{R}(\mathtt{Harry},\mathtt{JFK})\}\ \mathrm{R}(\mathtt{Ron},\mathtt{JFK})\ :-$$
$$\mathtt{F}(\mathtt{Ron},\mathtt{Harry}) \wedge \mathtt{U}(\mathtt{Ron},c) \wedge \mathtt{U}(\mathtt{Harry},c)$$

Earlier, a join was required in the body between F and U to ground the value of x. However, with the complete specification of friends, this join is now eliminated and the grounding step is faster. This leads to a marginal increase in performance, as shown in Figure 3.8.

Three-way coordination

The second experiments tests scalability in a slightly more complex scenario. We now generate triples of queries, corresponding to triangles in the social network structure, of the following form:

$$\{\texttt{R(Ron, IAH)}\}\ \texttt{R(Harry, IAH)} :-$$
$$\texttt{F(Harry, Ron)} \wedge \texttt{U(Harry,}c\texttt{)} \wedge \texttt{U(Ron,}c\texttt{)}$$
$$\{\texttt{R(Hermione, IAH)}\}\ \texttt{R(Ron, IAH)} :-$$
$$\texttt{F(Ron, Hermione)} \wedge \texttt{U(Ron,}c\texttt{)} \wedge \texttt{U(Hermione,}c\texttt{)}$$
$$\{\texttt{R(Harry, IAH)}\}\ \texttt{R(Hermione, IAH)} :-$$
$$\texttt{F(Hermione, Harry)} \wedge \texttt{U(Hermione,}c\texttt{)} \wedge \texttt{U(Hermione,}c\texttt{)}$$

We vary the size of the query set within the same parameters as before. As show in Figure 3.8, we observe a quadratic behavior of evaluation time in the number of queries. On profiling the results, we discover that the time to execute the MySQL queries grows only linearly and is negligible compared to the query matching time. We attribute the quadratic increase in time to complexity of unifier propagation in the matching phase. In our dataset, for three queries to match, the unifiers must propagate at least twice (as opposed to once when two queries are to match). Also, unifier propagation takes place for every pair of queries that have matched partially. For example, if the queries submitted by Harry and Ron match partially, they must wait for Hermione's query. Until Hermione's query arrives, Harry and Ron's queries may repeatedly propagate unifiers. However, despite the quadratic behavior of our system, the time to execute a query even in such a complex scenario is under 2ms.

Increasing the number of postconditions

The next set of experiments investigates the performance impact of an increase in the complexity of the coordination required. Specifically, we increase the number of postconditions per query, varying it from one to five. For each individual experimental run, all queries have the same number of postconditions. A sample set of three queries with two postconditions is given below.

$$
\begin{aligned}
&\{\texttt{R(Harry, SBN)} \wedge \texttt{R(Ron, SBN)}\}\ \texttt{R(Hermione, SBN)}\ :-\\
&\quad \texttt{F(Hermione, Harry)} \wedge \texttt{F(Hermione, Ron)} \wedge\\
&\quad \texttt{U(Ron,}\,c\texttt{)} \wedge \texttt{U(Hermione,}\,c\texttt{)} \wedge \texttt{U(Harry,}\,c\texttt{)}\\
&\{\texttt{R(Hermione, SBN)} \wedge \texttt{R(Ron, SBN)}\}\ \texttt{R(Harry, SBN)}\ :-\\
&\quad \texttt{F(Harry, Hermione)} \wedge \texttt{F(Harry, Ron)} \wedge\\
&\quad \texttt{U(Ron,}\,c\texttt{)} \wedge \texttt{U(Harry,}\,c\texttt{)} \wedge \texttt{U(Hermione,}\,c\texttt{)}\\
&\{\texttt{R(Hermione, SBN)} \wedge \texttt{R(Harry, SBN)}\}\ \texttt{R(Ron, SBN)}\ :-\\
&\quad \texttt{F(Ron, Hermione)} \wedge \texttt{F(Ron, Harry)} \wedge\\
&\quad \texttt{U(Harry,}\,c\texttt{)} \wedge \texttt{U(Ron,}\,c\texttt{)} \wedge \texttt{U(Hermione,}\,c\texttt{)}
\end{aligned}
$$

This represents a scenario where Hermione wants to travel with both her friends, Harry and Ron. Harry and Ron have analogous requirements. Note that this is different from the three way coordination mentioned above; cliques in the social graph are required for coordination, rather than just cycles. The intent of the coordination is that they all travel together from the same city to the same destination. Queries with a greater number of postconditions are generated in a similar fashion. Increasing the number of postconditions is associated with an increase in the number of queries that must be matched for successful coordination.

Figure 3.9 shows two components of the result obtained by executing 50000 queries. The first component corresponds to the time taken by the algorithm to find matching sets of queries, and the second part corresponds to the time taken by the MySQL database for query evaluation. We observe that both the matching time and the query execution time increase linearly with increasing number of postconditions, with the query execution time having a larger slope because of the increase in join size for the MySQL queries.

Stress-testing the query matching

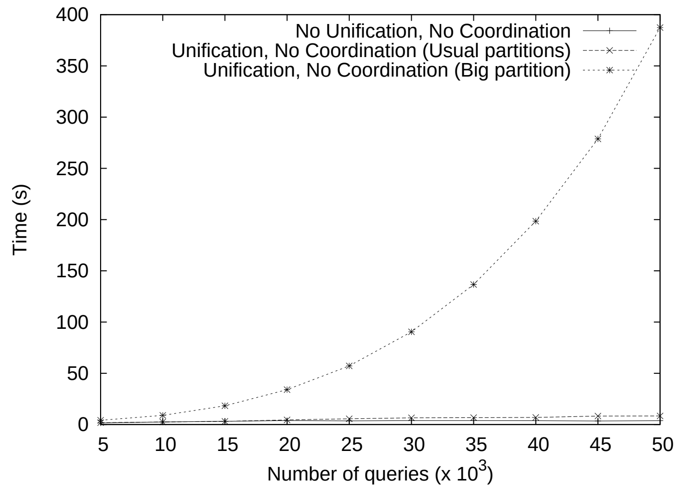


Figure 3.10: Scalability when queries do not match

Our next sets of experiments are designed to test the performance of query matching for workloads where little coordination can take place because most queries are unanswerable.

We first test this contingency using a query set generated to ensure that no query has a postcondition unifying with the head of another query. In this case, the unifiability graph does not have any edges; however, with the arrival of each query, index looks are performed to check for new edges. The unifier propagation phase of the algorithm is never initiated because postcondition and head atoms never unify. As expected the “no coordination, no unification” curve in

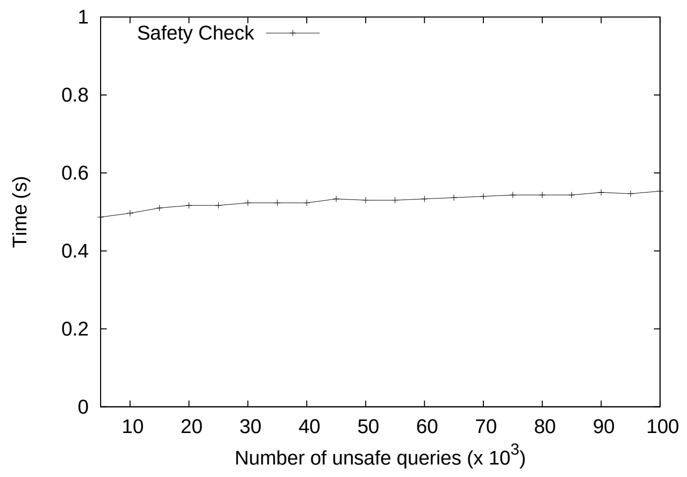


Figure 3.11: Evaluation time for safety check

Figure 3.10 is near-linear.

We also run experiments on a workload in which queries frequently have coordination partners but the system is never able to generate a single combined query in the evaluation phase. This process requires both graph construction and unifier propagation, and ideally the unifier propagation, even for queries without variables, should dominate the running time. If the matching algorithm was run after every query, one would initially expect the algorithm's running time to be at least quadratic.

As the "usual partitions" line in Figure 3.10 shows, the query evaluation time is nearly linear even though there is an increase in the number of pending queries (as no matching takes place) and many queries unify. In other words, the current set of queries forms a long chain in the unifiability graph but does not form cycles. After more careful analysis, we observe that the clustering in the social network graph restricts the size of partitions of our unifiability graph to a small number. This explains the high throughput in the experiment on the query set with high unification but no matching.

In order to stress test our system, we identify a big cluster in the social network graph and run experiments on this single large cluster. This change results in significant increase in the overall running time of our experiment, often not running to completion. We therefore run a set-at-a-time evaluation of such massively unifying partitions instead. Figure 3.10 shows the performance of such a process. The time to execute the queries is a quadratic curve. We observe that with increasing number of queries, the number of people trying to coordinate increases. For all of these queries, a big chain of potential matches is formed. For example, consider the first query. The unifier associated with this query must propagate; however, in every iteration of the system, the unifier propagates to only one more query. After iterating over every query for this particular unifier, the system determines that the matching fails. This repeated iteration of the order of the number of queries leads to the quadratic behavior that we observe. However, we conjecture that the execution time is still within reasonable bounds, given that thousands of people are trying to coordinate together. We therefore establish that for extremely huge coordinating groups, evaluating the queries set-at-a-time is definitely a better approach. By doing so, we wait till all coordination partners arrive before we actually run the algorithm.

Stress-testing the safety check

In the final experiment, we test the performance of the safety check. We load the system with twenty thousand queries that are unable to coordinate. Then, we add large sets of queries to the system. Such sets contain queries that will fail the safety check with respect to the queries already present in the system. We vary the size of such sets of queries from five to one hundred thousand. The results are shown in Figure 3.11. It clearly shows that the safety check does not add significant overhead to the system.

Discussion

In designing our experiments, our goal was not to design a full benchmark for entangled queries, but to understand whether this functionality is viable for use in a real-world system. As our results show, the algorithm is efficient in removing queries that are unable to be matched with others and queries that cause safety violations. The queries that are matched can be evaluated efficiently. The overall evaluation algorithm scales to workloads which are realistically sized with respect to today's social networks.

3.7 Future work

Notwithstanding the tractability bounds imposed by Theorem 3.2.3, a more expressive language for entangled queries would have many practical advantages. In this section, we present several concrete language extensions that would greatly enhance the usefulness of entangled queries. The syntax for entangled queries could be extended with features such as disjunction, union and aggregation in `WHERE` clauses. Consider a database that contains three tables: a table `Parties` with schema `(pid, pdate)`, a table `Friend` with schema `(name1, name2)`, and a relation `Attendance` with schema `(pid, name)`. Suppose a user named Harry wants to attend a party on Friday subject to the constraint that more than five of his friends attend this same party. This could be expressed as follows using aggregation:

```
SELECT party_id, 'Harry' INTO ANSWER Attendance
```

```
WHERE
  party_id IN (SELECT pid
               FROM Parties
               WHERE pdate='Friday')
AND
  (SELECT COUNT(*)
   FROM ANSWER Attendance A, Friend F
   WHERE party_id = A.pid AND
         A.name = F.name2 AND
         F.name1 = 'Harry') > 5
CHOOSE 1
```

"Soft" preferences, another possible extension of entangled queries, would allow coordination constraints to be relaxed when full coordination is difficult. For example, if Harry and Ron have trouble obtaining matching travel itineraries, they could instead request that their respective travel dates be as close together as possible.

It is also desirable to allow users to specify a ranking function on preferred query groundings. In our travel example, users who are coordinating on travel dates may prefer some dates to others. Disregarding their preferences may be acceptable if satisfying them precludes coordination, but the evaluation algorithm should favor coordinating sets $\mathcal{G}'$ that satisfy the users' preferences.

Finally, many applications could benefit from extended semantics that allow a query to return more than one answer tuple. Such semantics might allow users to request that *all* groundings of a query be included in the coordinating set, or that as many as possible be included up to some limit k. For instance, in a coordination-aware course enrollment system, students might request that they be enrolled in the same courses as their friends while the registrar ensures that no student enrolls in more than four courses.

Developing these and other extensions fully and designing suitable semantics and evaluation methods for them is ongoing work.

CHAPTER 4

ENTANGLED TRANSACTIONS

4.1 Introduction

Empty-handed I entered the world

Barefoot I leave it.

My coming, my going –

Two simple happenings

That got entangled. — Kozan Ichikyo.

In twentieth century data processing practice, programs and processes were largely solitary entities. Each operated individually to achieve a given task. Physical systems needed to handle multiple simultaneous processes, so the research community developed protection mechanisms to prevent interference. In the database community, this work culminated in the concept of a transaction. Such a classical transaction represents a discrete unit of data processing work as reflected in the ACID properties of atomicity, consistency, isolation, and durability: it provides the conceptual properties of being executed completely or not at all, of preserving database consistency as it runs, of running without interference from other transactions, and if committing, making its changes persistent.

However, in recent years data processing programs have become interdependent by design. For example, web services frequently interact and coordinate to carry out tasks spanning enterprises [13]. Coordination is now needed in domains ranging from course enrollment [36] and travel planning [22] to online social games such as Farmville, where gameplay is fundamentally collaborative. The coordination strategies used in these games are similar to those found in more "serious" application domains such as managing charity donations with gift matching [11] and auctions [45]. With Farmville now attracting over fifteen million users each day, data-driven coordination has become big business, and it is here to stay.

We recently started to take first steps towards the problem of supporting data-driven coordination [35, 22]. We introduced *entangled queries*, 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. However, most real-world data management applications that involve coordination require not just queries, but a transaction-like abstraction that covers larger units of work. As an example, assume that two friends, Mickey and Minnie, wish to travel to Los Angeles on the same flight and stay at the same hotel. Their arrival date is flexible, but their departure date is fixed. They start by jointly selecting a suitable flight. Once they know the flight number, and consequently their date of arrival in Los Angeles, they will try to make joint hotel reservations. With existing mechanisms, they can use entangled queries to coordinate on the choice of the flight and then on their choice of hotel. These queries, however, need to be embedded within a larger code unit that Mickey and Minnie separately execute and populate with their constraints such as the class of the hotel or airline restrictions. Once both their individual *entangled transactions* have been submitted, the system needs to match them up, execute the associated logic, and guarantee "transaction-like" semantics for this execution.

Research Challenges. What are these entangled transactions? How do they relate to entangled queries and to classical transactions? First, in order to define what we even mean by entangled transactions we need a clean semantic model which must capture both the fact that each entangled transaction represents a logical unit of work on its own, *and* that this work is dependent on input from other transactions in the system. Furthermore, the input from other transactions is not arbitrary; it is restricted to what can be achieved with entangled queries. This means entangled transactions have different semantics than nested transactions [44] and Sagas [19], where arbitrary communication is permitted between the components of a single unit of execution, or cooperative transaction groups [50], where such communication is regulated through complex custom policies. They are also different from split-transactions [54] as the components are defined statically and matched into a larger execution unit at runtime, and not the other way around.

The entangled transaction model must extend to transactions that contain more than one entangled query. Indeed, Mickey and Minnie's travel planning example requires entangled transactions with several entangled queries: the number of nights for the hotel reservation depends on the arrival date, which is not known until they have chosen a flight. This means Mickey and Minnie need to use separate entangled queries to coordinate on the flight and the hotel.

The semantics of classical transactions is closely tied to the ACID properties; it is appropriate to understand what analogues of these can be expected to hold for entangled transactions. For entangled transactions, isolation is clearly relaxed, but we also do not want to throw out the baby with the bathwater – that is, completely give up on the advantages and convenience of isolation between transactions. Our need to relax isolation is motivated by the novel semantics of entangled transactions, not by performance considerations as with relaxations of classical isolation [1]. Therefore, it appears isolation should be relaxed only "as far as necessary" to permit controlled communication through entangled queries.

Formalizing the above intuition is an interesting problem in its own right, but it is not sufficient for a full treatment of entangled isolation. It is also necessary to deal with the fact that when entangled transactions run, they see more of the system's state than classical transactions do. A transaction that receives an answer to an entangled query becomes aware of the existence of another entangled transaction in the system. Since the ultimate goal of isolation is to ensure that each transaction sees a consistent system state during execution, entangled isolation requires a consistent view of both the database and the concurrent processes.

Defining consistency preservation for entangled transactions is nontrivial. Intuitively, Mickey and Minnie's transactions still appear to be coherent units of work; neither one of them should individually introduce inconsistencies in the database if implemented and executed correctly. However, neither can execute by itself, so formalizing this intuition is not straightforward.

Even once a semantic model of entangled transactions is in place and we understand how the ACID properties extend to them, the details of a full *execution* model are far from obvious. Returning to Mickey and Minnie, suppose Minnie's transaction aborts after the two friends have chosen and booked a flight; the corrective action to be taken is not immediately clear. Also, it is likely that the two transactions may not arrive in the system simultaneously; if one of them has to wait for the other, it is important to ensure usability of the system by other transactions in the interim. Designing an execution model to handle issues like the above in a principled way raises many research questions.

Last but not least, entangled queries are not useful until they are supported in a real system that can be deployed in practice. Designing the architecture of such a system and combining it with existing DBMS functionality presents deep systems challenges.

Our contributions. In this paper, we lead the reader into the new world of entanglement. First, we review our building block of entangled queries (Section 4.2). We then introduce a model of entangled transactions that comes with analogues of the classical ACID properties. Our model permits trading off isolation to achieve greater concurrency, albeit at the cost of some loss of consistency, resulting in the definition of isolation levels for entangled transactions (Section 4.3). Second, we discuss execution models for entangled transactions. We outline the major design issues involved and present a specific model that we found especially suitable for our motivating application scenarios (Section 4.5). Third, we outline the challenges that arise when implementing a system supporting entangled transactions. We present the architecture of our prototype implementation of entangled transactions within the Youtopia system (Section 4.6). The prototype is implemented at the middle tier, and as such can be used with any existing DBMS. Experiments with our prototype show that the overheads associated with supporting entangled transactions are acceptable for real-world use.

4.2 Entangled queries

Entangled queries are expressed in extended SQL as follows:

```
SELECT select_expr
INTO ANSWER tbl_name [, ANSWER tbl_name] ...
[WHERE where_answer_condition]
CHOOSE 1
```

The `WHERE` clause is a standard condition clause that may refer to both database and `ANSWER` relations. The `ANSWER` relations are not database tables;

they serve only as names that are shared among queries and permit entanglement. As in our previous work [22], the `WHERE`-clause is restricted to contain only select-project-join queries.

To continue with our example from the introduction, suppose Mickey wants to travel to Los Angeles on the same flight as Minnie. He can express this with the entangled query below.

```
SELECT 'Mickey', fno, fdate INTO ANSWER Reservation
WHERE fno, fdate IN
    (SELECT fno, fdate FROM Flights
     WHERE dest='LA')
AND ('Minnie', fno, fdate) IN ANSWER Reservation
CHOOSE 1
```

The name `Reservation` refers to a conceptual relation which collects the answers to all the queries relating to flight bookings. The `SELECT` clause specifies Mickey's own expected answer, or, in other words, his contribution to the answer relation `Reservation`. This is a tuple containing the constant `Mickey`, the flight number and the date of the booking. The existence of Mickey's answer, however, is conditional on two requirements, which are given in the `WHERE` clause. First, the flight's destination must be Los Angeles. Second, the answer relation must also contain a tuple with the same flight number and date but `Minnie` as the passenger name. The `CHOOSE 1` at the end of the query specifies that the system should choose only one flight, even if more than one might be suitable.

Now suppose Minnie actually wants to fly with Mickey, but she wants to fly only on United. Her query is as follows:

```
SELECT 'Minnie', fno, fdate INTO ANSWER Reservation
```

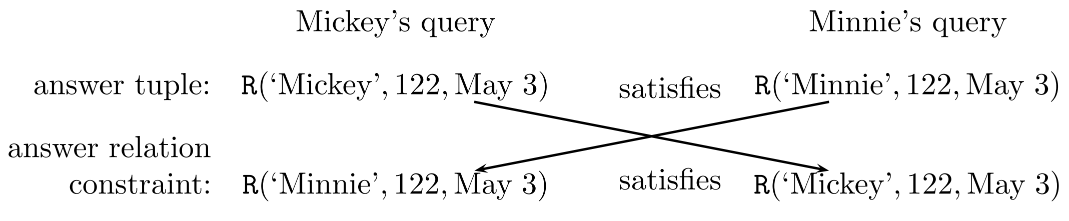
Flights

fno	fdate	dest
122	May 3	LA
123	May 4	LA
124	May 3	LA
235	May 5	Paris

Airlines

fno	airline
122	United
123	United
124	USAir
235	Delta

(a)



(b)

Figure 4.1: (a) Flight database (b) Mutual constraint satisfaction

```
WHERE fno, fdate IN
    (SELECT fno, fdate
     FROM Flights F, Airlines A WHERE
          F.dest='LA' and F.fno = A.fno
          AND A.airline = 'United' )
AND ('Mickey', fno, fdate) IN ANSWER Reservation
CHOOSE 1
```

When the system receives the two queries, it answers both of them simultaneously in a way that ensures a coordinated choice of flight. If the database is as shown in Figure 4.1 (a), the system nondeterministically chooses either flight 122 or 123 and returns appropriate answer tuples. Figure 4.1 (b) shows the mutual constraint satisfaction that takes place in answering for 122; the relation name `Reservation` is abbreviated as `R`. Neither Mickey nor Minnie sees the other's answer, but each of them is guaranteed that all answer constraints have been met.

After Mickey and Minnie receive answers to their queries, each of them can book a seat on the flight and date specified. The above queries are simplified; in practice, they would perform more work such as verification of seat availability.

Section 3.2 gives an overview of the semantics of entangled queries; for more details, see our previous work [22]. The semantics makes use of the notion of a *grounding* for each entangled query. To compute a grounding essentially means to evaluate of the portion of the `WHERE` clause which does not refer to an `ANSWER` relation. This identifies the set of acceptable answers for each individual query; for Minnie's query, for example, it would identify that only answers involving flights 122 or 123 are suitable. Answering a *set* of entangled queries involves choosing an acceptable answer for each individual query such that the corresponding individual answers all satisfy the appropriate constraints.

4.3 Entangled Transactions

In this section, we introduce our model for entangled transactions and discuss the new meaning of the ACID properties in the presence of entanglement.

4.3.1 Syntax and Semantics

Entangled transactions have the following syntax.

```
BEGIN TRANSACTION [WITH TIMEOUT duration]
[SQL standard syntax | entangled_query | ROLLBACK]*
entangled_query
[SQL standard syntax | entangled_query | ROLLBACK]*
COMMIT
```

```
BEGIN TRANSACTION WITH TIMEOUT 2 DAYS;

SELECT 'Mickey', fno, fdate AS @ArrivalDay
INTO ANSWER FlightRes
WHERE fno, date IN
    (SELECT fno, fdate FROM Flights
     WHERE dest='LA')
AND ('Minnie', fno, fdate) IN ANSWER FlightRes
CHOOSE 1;

-- (Code to perform flight booking omitted)

SET @StayLength = '2011-05-06' - @ArrivalDay;

SELECT 'Mickey', hid, @ArrivalDay, @StayLength
INTO ANSWER HotelRes
WHERE hid IN
    (SELECT hid FROM Hotels
     WHERE location='LA')
AND ('Minnie', hid, @ArrivalDay, @StayLength) IN
ANSWER HotelRes
CHOOSE 1;

-- (Code to perform hotel booking omitted)

COMMIT;
```

Figure 4.2: Example Entangled Transaction

Figure 4.2 shows a transaction that Mickey might run to coordinate with Minnie on a flight and a hotel in Los Angeles, as discussed in the introduction. The table `Hotels` contains information about hotels, including a hotel id (`hid`) and location attribute. `FlightRes` and `HotelRes` are the answer relations for flight and hotel booking coordination, respectively. `HotelRes` has attributes for customer name, hotel id, arrival date, and number of nights.

An entangled transaction is specified by the code enclosed within `BEGIN TRANSACTION` and `COMMIT`. In addition to the functionality offered by an ordinary transaction, an entangled transaction also contains one or more entangled queries. Calls to evaluate an entangled query are blocking: the transaction does not proceed until the entangled query receives an answer. The programmer may directly bind the values returned by an entangled query to host variables by specifying `AS @varname` next to the appropriate value in the query; this can be seen in the example above with `@ArrivalDay`.

Because of the blocking calls to evaluate entangled queries, we associate a `timeout` parameter with each entangled transaction. This parameter limits the maximum time that this transaction may "wait" in the system for its entanglement partner(s). If a particular entangled query within the transaction is unable to succeed before the timeout expires, then the entire transaction is unable to complete. An error is thrown and must be handled by the application code. Entangled query failure is a relatively complex phenomenon that can happen for several reasons, not just the absence of a partner. Section 3.2 in the Appendix contains a more in-depth discussion of this issue and how it impacts transaction execution.

From a programmer's perspective, entangled queries have an additional advantage beyond allowing information exchange: They provide explicit synchronization points between transactions. This can be useful if the programmer knows the code of other transactions in the system. Once an entangled query is answered, a transaction can assume that all entanglement partners have executed the code preceding their corresponding entangled queries. For instance, if Minnie manages to coordinate with Mickey's transaction on a hotel, she knows that he has already booked his flight.

4.3.2 Consistency

We now present extensions of the ACID properties to entangled transactions. Consistency and isolation are particularly affected by entanglement, so we start by treating each of them in turn.

Classically, consistency is an abstract property of databases which transactions preserve by the following assumption:

Assumption 4.3.1 (Consistency)**.** *Every transaction, if executed by itself on an initially consistent database, will produce another consistent database.*

The motivation behind this assumption is that an individual transaction is a logical and self-contained unit of work. A correct implementation of such a transaction will never deliberately create data inconsistencies, except perhaps temporarily in the middle of its execution. The only time consistency of the final database is not guaranteed is if the initial database was inconsistent as well.

Assumption 4.3.1 is used to infer global consistency guarantees for the execution of a *set* of transactions. Suppose the permissible concurrent executions are constrained in such a way that every individual transaction sees (i.e. reads) a consistent version of the database as it runs. Then, the above assumption allows us to infer that any set of concurrent transactions, run on an initially consistent database, will produce another consistent database.

To formulate an analogous guarantee for entangled transactions, we need an equivalent of Assumption 4.3.1. The key is deciding what constitutes a logical and self-contained unit of work; this is non-trivial for an entangled transaction as it cannot execute by itself.

Three candidates for units of work are individual entangled transactions, *groups* of transactions that entangle during execution, and non-entangled *portions* of individual entangled transactions. With respect to the example shown in Figure 4.2, the first option would correspond to Mickey's transaction, the second to Mickey *and* Minnie's transactions, and the third to the two non-entangled code segments that are executed between entangled queries. The first option maintains the closest correspondence between entangled transactions and classical transactions; it is the one we use in this paper, and we leave the others as future work.

It is not obvious what it means for an individual entangled transaction to constitute a unit of work, given that a transaction like the one in Figure 4.2 is unable to run and complete by itself. However, intuitively, the only information this transaction needs "from the outside" is answers to the two entangled queries so that it knows which flight and which hotel to book. As long as Mickey's transaction is executed in the system alongside *some* process that provides this information, it will be able to complete correctly. This other process could be Minnie's transaction, but it could also in principle be a "query answering oracle" whose only functionality is to create Mickey's answer tuples.

We formalize the notion of an entangled query oracle as follows.

Definition 4.3.2 (Entangled Query Oracle)**.** *An entangled query oracle O is a process that executes alongside an entangled transaction t. Whenever t poses an entangled query, the oracle generates an answer and returns it to t. The oracle has no direct effect on the database's state, i.e. it performs no writes.*

The above definition deliberately does not constrain the kinds of answers that the oracle may supply to t. However, these answers should clearly simulate those received in true entanglement.

Definition 4.3.3 (Valid oracle answer)**.** *Suppose a transaction t executes with an oracle O and poses an entangled query q at a time when the state of the database is D. An answer to q returned by the oracle is valid if it directly corresponds to a grounding of q on D.*

Definition 4.3.4 (Valid oracle execution)**.** *Suppose a transaction executes alongside* *an oracle O. If the oracle returns a valid answer to each entangled query, the entire execution is* valid.

This allows us to formulate the following consistency assumption for entangled transactions.

Assumption 4.3.5 (Oracle Consistency)**.** *Suppose an entangled transaction executes by itself on an initially consistent database, using an entangled query oracle to obtain answers to the entangled queries it poses, and suppose the execution is valid. Then the execution will produce another consistent database.*

This assumption is close in spirit to Assumption 4.3.1. It states that an entangled transaction will produce consistent "output" – i.e. a set of database writes that together do not violate consistency – as long as it is presented with consistent "input" – i.e. a consistent view of the data and valid answers to its entangled queries. This assumption holds for Mickey's transaction and is likely to hold for typical transactions in most realistic settings.

As with classical transactions, Assumption 4.3.5 can be used to infer a consistency guarantee for the execution of a set of entangled transactions. To this end, we again need to constrain the permissible concurrent executions so that each transaction is guaranteed to receive consistent "input". That is, we need to define isolation for entangled transactions, and it is to this issue that we turn next.

4.3.3 Isolation

Classical isolation is motivated by the need to provide each transaction with a consistent view of the database as it runs. As discussed, this together with

Assumption 4.3.1 guarantees that the final database produced by a set of concurrent transactions is consistent.

An elegant way to define classical isolation is in terms of *serializability*, i.e. equivalence of an execution schedule to a serial execution schedule with the same transactions. In a serial schedule, Assumption 4.3.1 guarantees that each transaction does indeed see a consistent view of the database, so serial execution is a suitable gold standard for consistency. Equivalently, classical isolation can be defined as the avoidance of certain execution anomalies such as dirty reads and unrepeatable reads [1].

For entangled transactions, serializability is not directly applicable. However, we can use our entangled query oracles to define *oracle-serializability*, that is, equivalence to a schedule where the entangled transactions execute serially alongside an oracle. We can also formulate an anomaly based definition of isolation based on the classical anomalies and some new entanglement-related ones.

We have developed both an anomaly based and an oracle-serializability based definition of entangled isolation and proved a theorem that relates them to each other. Due to space constraints, we give only a high-level overview of each of these contributions here; the full formal treatment is found in the Appendix, Section 4.4.

Anomaly-based definition

We first outline our anomaly-based definition of entangled isolation. Anomalies are pathological phenomena where a transaction observes an inconsistent view of the system state as it runs. If no such anomalies occur during execution, then by Assumption 4.3.5, we know that the final database produced after the execution of a set of entangled transactions is consistent. We begin by introducing

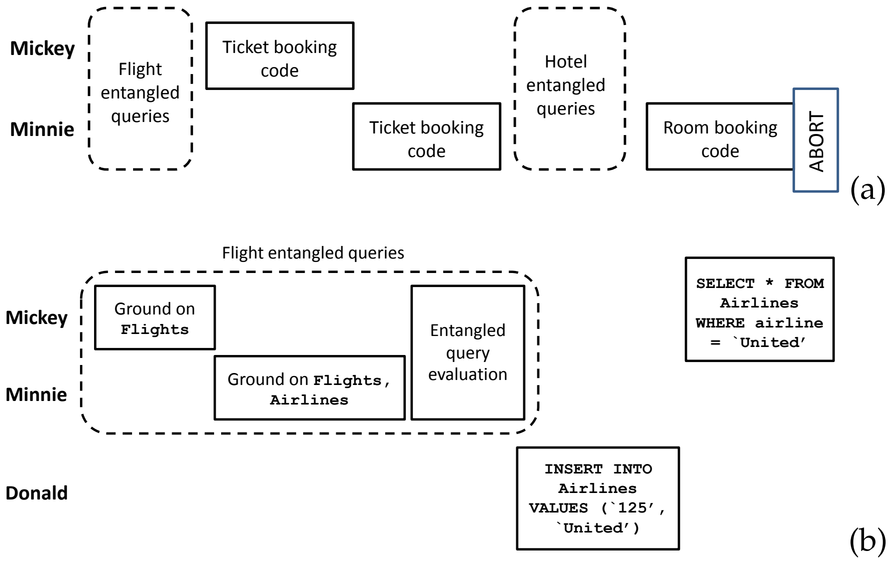


Figure 4.3: (a) Widowed transaction (b) Unrepeatable quasi-read

the anomalies which are unique to entangled transactions. Classical anomalies such as dirty reads and unrepeatable reads can also happen in the entangled setting and must be avoided.

Widowed transactions To execute correctly and preserve consistency, entangled transactions need more than just a consistent view of the database. They need a consistent view of the entire system state, insofar as they have access to it. For classical transactions, the only accessible system state is the database. However, an entangled transaction *t* that has received an answer to an entangled query knows something else: it knows that there is another process running alongside it. If this other process subsequently aborts, *t*'s view of the system may become inconsistent.

Consider for example the scenario in Figure 4.3 a), where Mickey runs the transaction in Figure 4.2 and Minnie runs a symmetric transaction containing the query from Section 4.2. The transactions successfully entangle on both the flight and hotel queries. However, while performing the hotel booking, Minnie's transaction aborts. Mickey's transaction may still in principle be able to complete its hotel booking; however, Mickey would be booking a room based on the assumption that Minnie is traveling with him, and this may no longer be true. Mickey's transaction is *widowed* due to the abort of its entanglement partner. The **widowed transaction anomaly** is our first isolation anomaly which is unique to the entangled setting.

The additional system state visible to entangled transactions can be made explicit using the ANSWER relations. For simplicity, assume there is just one such relation. The transactions perform operations on this relation during entangled query answering. The answering process starts with a set of simultaneous *writes* by all transactions to the ANSWER relation – each transaction writes its corresponding answer tuple. Next, each transaction receives a guarantee that its partner's answer tuple is in the ANSWER relation – that is, the transaction performs an implicit *read* on the ANSWER relation. The reads are again performed simultaneously by all transactions.

The above discussion makes it clear why widowed transactions are a problem: two transactions that entangle perform a dirty read of each other's writes to the ANSWER relation. If one later aborts, the other's view of the ANSWER relation becomes inconsistent.

Unrepeatable quasi-reads The second new class of isolation anomaly is associated with the entangled query answering process itself, or more precisely with the computation of *groundings* for the queries. As explained in Section 4.2, the evaluation of a set of entangled queries conceptually begins by *grounding* each query. The actual evaluation algorithm does not need to operate in this way, but groundings are a useful tool for analysis. A grounding is a read on the database that corresponds to the portion of the WHERE clause of an entangled query that does not refer to any ANSWER relation. Two queries that entangle may ground on the same data; for example, Mickey and Minnie's flight entangled queries both ground on the `Flights` table by selecting all flights to Los Angeles.

If groundings are not handled carefully, anomalies can occur due to interference. To see an example of this, start with Mickey and Minnie's queries from Section 4.2 and consider the execution schedule shown in Figure 4.3 b). Minnie's query grounds on both `Flights` and `Airlines`, whereas Mickey's grounds only on `Flights`. Mickey receives flight number 122 as an answer. He decides to check which flights are operated by United, to gain a better understanding of the options from which his answer was chosen. However, between the time that Mickey gets the entangled query answer and the time he reads `Airlines` himself, Donald adds a new flight with number 125 operated by United. Clearly this creates a problem for Mickey. Mickey does not perform a classical unrepeatable read, because he only reads `Airlines` once. The key to understanding this anomaly is that Minnie has read the same table *prior* to entanglement; intuitively, there has been some information flow from the `Airlines` table to Mickey's transaction during the answering of his entangled query. His subsequent explicit read of `Airlines` therefore shows him information that is inconsistent with his prior knowledge of this relation.

This and other similar anomalies can be formalized using the notion of a *quasi-read*, which models the information flow that occurs through entangled query answering. In our example, we say that Minnie's grounding read on `Airlines` was also a *quasi-read* by Mickey's transaction on the same table. It is now clear that Mickey has indeed performed an unrepeatable read on `Airlines`: a quasi-read before Donald's write and a normal read afterwards. Consequently, we introduce **unrepeatable quasi-reads** as the second class of anomalies which is unique to entangled transactions. This includes all anomalies involving two reads on the same object by the same transaction, at least one of which is a quasi-read, and where the object changes value between the reads.

Entangled isolation Our anomaly-based definition of entangled isolation prohibits widowed transactions and unrepeatable quasi-reads, as well as all classical isolation anomalies; that is, an execution schedule is entangled-isolated if it exhibits none of these anomalies. The definition is presented formally in the Appendix, Section 4.4.2; it uses the notion of a *conflict graph* which tracks operation conflicts between transactions to exclude both the classical anomalies and unrepeatable quasi-reads. As in the classical case, it is possible to relax this definition to admit lower isolation levels by permitting a specific subset of the above anomalies to occur.

Oracle-serializability based definition

The key idea behind oracle-serializability is to compare a given execution schedule to a schedule where the same entangled transactions are executed serially alongside a suitable query oracle. The oracle answers need to be consistent across transactions; if Mickey's transaction executes first and receives 122 as the answer to its flight query, Minnie's transaction should also receive 122 as an answer to the corresponding query when it executes later.

As explained in Section 4.4.3 in the Appendix, the oracle is constructed in a custom way for each schedule σ. It essentially "stores" the entangled query answers that each transaction received in σ and returns them verbatim at the appropriate point during serial execution, whether or not these answers are valid (as per Definition 4.3.3).

We define a schedule σ to be oracle-serializable if there is some serial order of the transactions in σ for which execution with the above oracle is valid and yields the same final database as σ when run on the same starting database. This definition is the entangled equivalent of classical final-state serializability.

The following theorem is our main result and relates both of our definitions of isolation. It is proved in the Appendix, Section 4.4.4.

Theorem 4.3.6. *Any schedule that is entangled-isolated is also oracle-serializable.*

As expected, the serialization order for the oracle execution must be consistent with the conflict graph.

Enforcing Isolation

To enforce isolation for entangled transactions, a system must prevent widowed transactions and unrepeatable quasi-reads, in addition to the classical isolation anomalies. Widowed transactions can be avoided by enforcing **group commits**: if two transactions entangle, both must either commit or abort. This pairwise requirement induces a requirement on groups of transactions that have entangled with each other directly or transitively: all transactions in such a group must either commit or abort. As for repeatability of quasi-reads, it can be enforced for example using an appropriate locking protocol. In a system that uses Strict 2PL, Donald's write in Figure 4.3 b) would not have been possible, as Minnie's transaction would have held a read lock on the `Airlines` table until commit.

4.3.4 Atomicity and Durability

Because we have identified individual entangled transactions as our basic unit of work in the system, we can define atomicity and durability based on their classical equivalents. For atomicity, each entangled transaction must execute to completion or be rolled back. For durability, if an entangled transaction commits, its database writes must be persistent despite any system failures.

The above are the minimal atomicity and durability requirements which the system must enforce at all times. Stronger guarantees are sometimes possible. For example, at isolation levels that disallow widowed transactions, all *groups* of transactions that entangle together are guaranteed to execute atomically. Moreover, once a transaction commits, both its changes and the changes of all its entanglement partners are durable.

4.4 Formalizing isolation

In this section, we formalize the presentation of entangled isolation from Section 4.3.3. Our presentation does not handle predicate reads explicitly, nor does it deal with schedules produced in systems using explicit data versioning. The entire discussion that follows can be extended to handle predicate reads with suitable additional formalism [1]. We have chosen not to present this extension here as it requires significant additional notation and is orthogonal to our main focus, which is the unique meaning of isolation for entangled transactions and specifically the differences between entangled and classical isolation. Multiversion settings come with their own unique challenges – as in the case of classical transactions – and we leave their treatment as future work.

4.4.1 Transaction schedules

Operations A schedule for entangled transactions is very similar to a schedule for ordinary transactions and contains the familiar read, write, abort and commit operations, denoted R, W, A and C respectively. The only two differences pertain to entangled query processing: certain reads are distinguished as *grounding* reads and the schedules make use of an additional operation – *entan-*

glement.

Entangled query processing begins by grounding the queries. This can be done individually or through a combined query as in [22]. To remain implementation-independent, we model the more general case where each query grounds separately. Each grounding is a set of reads; we distinguish these as grounding reads and denote them as R^G rather than just R. Technically, the grounding reads are not performed by the transaction itself, but by the system *on behalf* of the transaction. Nonetheless, they clearly represent information flow from the database into the transaction; we therefore associate grounding reads with the transaction posing the entangled query, rather than with a special "system" process.

The next step in entangled query evaluation is to find a set of groundings that satisfy each other's postconditions, i.e. to construct the answer relation. We model this with an explicit *entanglement* operation, denoted E. We assign each entanglement operation a unique identifier, and associate each entanglement operation with the set of identifiers of the transactions that receive answers, denoted as $\mathcal{T}_k$, where k is the identifier of the entanglement operation. To introduce notation by example, if transactions 1 and 3 participate in entanglement operation 7, this is denoted as $E^7_{1,3}$

Validity constraints An entangled transaction execution schedule is a sequence of read, write, entangle, commit and abort operations. Obviously it is possible to construct sequences of such operations that do not correspond to any possible real transaction execution. This means that we need a notion of *valid* schedules – sequences of operations that are constrained to match the semantics of entangled transactions. The constraints involved are straightforward; however, for completeness we present them next.

First, for every transaction i, a valid schedule may contain at most one of $\{A_i, C_i\}$. Indeed, we find it helpful to require that it contain *exactly* one of these, thus ensuring we work with complete schedules (histories) only. Second, for every transaction i that aborts or commits, the abort or commit operation must be the last operation by i. Third, if a transaction i performs a grounding read $R_i^G(x)$ on some object x, then the schedule must contain either a subsequent entanglement operation involving i or a subsequent A_i. Fourth, consider the interval between a grounding read by transaction i and the next entanglement or abort operation by i that follows it (such an operation must exist by the previous requirement). During that interval, i may not perform any operations other than additional grounding reads. This is because entangled query evaluation calls are blocking: i cannot proceed with subsequent reads or writes until entanglement occurs and it receives answers.

Schedules We can now formally define (valid) schedules for entangled transactions.

Definition 4.4.1 (Schedules)**.** *A schedule is a sequence of the following operations: read, write, abort, commit, and entangle, where each operation is tagged with one or more transaction identifiers, and the sequence satisfies the validity constraints listed above.*

An example schedule is as follows:

$$R_1^G(x)R_2^G(y)R_3(z)E_{1,2}^1W_1(z)W_2(w)C_1C_2C_3$$

In this schedule, transaction 1 grounds on x, then 2 grounds on y. 3 performs a normal read on z, after which 1 and 2 and entangle together based on their grounding reads. Finally, 1 and 2 perform a write to z and w respectively. All three transactions commit.

When a schedule σ is executed on a database, the final database produced reflects exactly the writes of all the *committed* transactions in σ, in the order in which these writes occurred in σ. We assume that the entangled query evaluation algorithm is deterministic, i.e. always returns the same answers when processing the same set of queries on the same database. This implies that whenever σ runs on the same starting database, it produces the same final database. Lifting this assumption is possible but would make the presentation that follows more complex.

4.4.2 Anomaly-based entangled isolation

We now formalize our anomaly-based isolation definition.

Preliminaries

Quasi-reads Suppose two transactions i and j perform grounding reads on two different objects, say x and y respectively, and entangle immediately afterwards. Although i has not directly read y and j has not directly read x, there has been some information flow from each object to each transaction through entanglement. As discussed, we model this information flow through *quasi-reads*. Whenever a transaction performs a grounding read on an object, all of its partners in the subsequent entanglement operation are considered to perform a simultaneous quasi-read on the same object. We denote a quasi-read by R^Q. In the pathological case where a transaction performs a grounding read but there is no subsequent entanglement operation (i.e. the transaction aborts instead), no quasi-reads are associated with that grounding read.

Given an entangled transaction schedule, it is straightforward to identify which grounding reads are associated with quasi-reads by other transactions and to add in the quasi-reads explicitly. Concretely, our example schedule can be rewritten as follows:

$$\Big(R_1^G(x)R_2^Q(x)\Big)\Big(R_2^G(y)R_1^Q(y)\Big)R_3(z)E_{1,2}^1W_1(z)W_2(w)C_1C_2C_3$$

The brackets surrounding a set of operations denote that the operations occur simultaneously. Often they will not be necessary as the timing of the quasi-reads will be clear.

In the remainder of this section, we use the word *schedule* to refer to valid entangled transaction schedules in which the quasi-reads are made explicit. The unqualified term *read* refers to any read operation including a grounding read or quasi-read.

Conflicts Given a schedule σ, we can compute a *conflict graph* for the *committed* transactions in σ. This is a graph where the nodes correspond to transaction identifiers and edges are added based on conflicting operation pairs on the same object. A pair of operations on the same object by two different transactions i and j are *conflicting* if at least one is a write. If the operation by i occurs in the schedule first, we add an edge from i to j in the conflict graph.

It is important to realize that the graph is defined only for those transactions that *commit*; we only place restrictions on anomalies that affect committed transactions. This allows reasoning about schedules that exhibit correct isolation, but could not have been generated by preventative (pessimistic) concurrency control implementations. For a further discussion of this issue, see [1].

Entangled isolation

The following three requirements on schedules σ can be used to rule out isolation anomalies.

Requirement 4.4.2 (No cycles). *The conflict graph for σ must be acyclic.*

Requirement 4.4.3 (No read-from-aborted). *If i is a transaction that aborts and j a transaction that commits, σ may not contain the sequence of operations $W_i(x) \ldots R_j(x)$*

Requirement 4.4.4 (No widowed transactions). *If σ contains an entanglement operation associated with transactions i and j, then it may not contain both A_i and C_j.*

Requirements 4.4.2 and 4.4.3 are sufficient to rule out classical isolation anomalies and unrepeatable quasi-reads, as the latter are made explicit in the schedule. Note also that when two transactions ground on the same object and entangle based on that grounding, Requirement 4.4.2 guarantees that they see the same version of this object; as explained in Section 3.2, this is necessary for correct entangled query answering. If the transactions were to ground on different versions of the same object and entangle, this would be an instance of an unrepeatable quasi-read.

We now formally define entangled isolation.

Definition 4.4.5 (Entangled isolation). *A schedule is entangled-isolated if it satisfies Requirements 4.4.2, 4.4.3 and 4.4.4.*

4.4.3 Oracle-serializability

Classical serializability compares a given execution schedule to a schedule where the same transactions execute serially. We formulate an entangled analogue of this where each transaction executes alongside an oracle. We can then reason about equivalence between an entangled schedule and its *oracle-serializations*.

Oracle construction

Suppose we are given a schedule σ; we explain how to construct an oracle O_σ that enables serial execution of the transactions in σ on a given starting database D. The oracle is customized to σ and to D, but not to any serialization order of the transactions in σ.

To build the oracle, identify all the entanglement operations in σ and create a procedure in the oracle that corresponds to each of these operations. In any serial schedule involving the transactions in σ, this entanglement operation will correspond to a number of oracle calls by the individual transactions, and the appropriate oracle procedure will be invoked each time. For example, suppose transactions i and j entangle in an operation E^k_{ij}, and the entangled queries involved were q_i and q_j. The oracle contains a procedure specific to E^k_{ij}. This procedure will be invoked twice in any serial execution – once when i executes and poses q_i, and once when j executes and poses q_j.

The procedure to handle an entanglement operation E^k is as follows. By observing σ's execution on D, we can keep track of the actual answers returned at each entanglement operation E^k. The answers can be recorded in a data structure Ans_k which is a map from $\mathcal{T}_k$ to the set of answers, so that $Ans_k(i)$ is the answer entanglement operation k returns to transaction i. The oracle makes use of the answer set Ans_k directly; when answering the query posed by transaction i, it simply returns $Ans_k(i)$. Therefore, by construction, it is guaranteed that the oracle returns consistent answers to all corresponding entangled queries, as the answers in Ans_k are assumed to be consistent. On the other hand, the oracle answers are not guaranteed to be valid according to Definition 4.3.3. This means that invalid executions (Definition 4.3.4) for some transactions may be possible with O_σ.

Oracle-serializations of schedules

We now define an oracle-serialization of a schedule σ.

Definition 4.4.6 (Oracle-serialization)**.** *Let σ be an entangled schedule run on a starting database D and O_σ the entangled oracle for σ and D as described above. An oracle-serialization of σ on D is a schedule generated when the* committed *transactions in σ are totally ordered in some way and each transaction executes individually alongside O_σ in this order. We use $os(\sigma)$ to denote an oracle-serialization of σ.*

Oracle-serializations include only the committed transactions in σ; this is consistent with our formalization of entangled isolation. Note also that oracle-serializations of σ will in general not contain the exact same operations as σ itself. Specifically, the entangled transactions no longer perform grounding reads or quasi-reads. For instance, consider our entangled schedule example from before:

$$R_1^G(x)R_2^Q(x)R_2^G(y)R_1^Q(y)R_3(z)E_{1,2}^1W_1(z)W_2(w)C_1C_2C_3$$

Suppose we serialize this schedule in the order $3, 1, 2$ on some database D. The serialization is as shown below; O_l^m denotes an oracle call by transaction l with the same entangled query that it posed in entanglement operation m in σ. We have not listed the specific answers returned by the oracle, so the below can more correctly be considered a *template* for an oracle-serialization of σ.

$$R_3(z)C_3O_1^1W_1(z)C_1O_2^1W_2(w)C_2$$

The grounding reads for the entangled queries posed by transactions 1 and 2 are no longer there. This is unsurprising, as the schedule represents reads and writes *to the database*, whereas now the entangled queries are answered without any reference to the database, solely based on the set of answers stored in the oracle.

Oracle-serializability

We now define oracle-serializability – the analogue of final-state serializability for entangled transactions.

Definition 4.4.7 (Oracle-serializability)**.** *An entangled schedule σ is oracle-serializable if there is some serialization order of the transactions in σ such that for all starting databases D, the oracle-serialization of σ in that order on D is a valid execution and yields the same final database as σ.*

Note that although the oracle required for serialization depends on the starting database the serialization order does *not*.

4.4.4 Entangled isolation guarantees

In this section, we give the proof of Theorem 4.3.6, that is, we argue that any schedule which is entangled-isolated is also oracle-serializable.

Proof. Start with any entangled-isolated schedule σ and compute its conflict graph. Choose any total ordering of the transactions in σ consistent with a topological sort on the graph; such an ordering must exist as the graph is acyclic by Requirement 4.4.2. Choose an arbitrary D and let $os(\sigma)$ be the oracle-serialization of σ on D with respect to that order. We must show that:

(1) the execution of $os(\sigma)$ on D is valid, and

(2) the final database produced is the same as that produced by σ itself executing on D.

To prove (1), we need to show that at the time the oracle returns $Ans_k(i)$ to transaction i, the state of the database is such that $Ans_k(i)$ is valid. To do so, we introduce a technical device we call *validating reads*. Intuitively, suppose some process were to monitor the execution of $os(\sigma)$ and actually ground each entangled query before the oracle answers it, in order to perform a validity check. For the purpose of this proof, we explicitly introduce such validating reads into the schedule $os(\sigma)$ and associate them with the transaction that asked the original entangled query. Consider our example oracle serialization from above:

$$R_3(z)C_3O_1^1W_1(z)C_1O_2^1W_2(w)C_2$$

With validating reads (denoted as R^V) added, this becomes

$$R_3(z)C_3R_1^V(x)O_1^1W_1(z)C_1R_2^V(y)O_2^1W_2(w)C_2$$

For every validating read in $os(\sigma)$, there is a grounding read in σ by the same transaction on the same object, and vice versa. In fact, suppose we could show that every validating read in $os(\sigma)$ sees the same value as the corresponding grounding read in σ. Then we could guarantee that all oracle answers are valid and point (1) follows. This is because the oracle answers are exactly the answers in Ans_k, and these were computed based on the result of the actual grounding reads in σ.

Before we prove the above statement about validating and grounding reads, consider how we might prove point (2). Suppose we add a dummy transaction at the end of both σ and $os(\sigma)$ that reads every object mentioned in σ. It suffices to show that this transaction reads the same values in both schedules.

We show both (1) and (2) by proving the stronger statement that every read in $os(\sigma)$, including validating reads and reads by the dummy transaction, sees the same value as the corresponding read in σ. We prove this using an inductive argument similar to that used for classical conflict-serializability. We make (an entangled equivalent of) the standard determinism assumption – if a transaction sees the same values for its reads and entangled query answers, and if the process that provides the entangled query answers does not abort, then the transaction will produce the same writes.

The first transaction in the serialization order in $os(\sigma)$ sees the same values as it did in σ, because in σ its reads depended only on the original values and the results of its own writes; otherwise, it would have had an incoming edge in the conflict graph and could not have been serialized first. It also receives the same entangled query answers as it did in σ, by construction. Finally, in both schedules, it has an entanglement "partner" that does not abort – a real transaction in σ that commits due to requirement 4.4.4, and the oracle in $os(\sigma)$. Consequently, by the determinism assumption, it produces the same writes as before. As our inductive step, consider the nth transaction in $os(\sigma)$, and suppose it reads object x. In σ, this transaction cannot have seen writes to x from any aborted transactions, since that would violate Requirement 4.4.3. Consider all committed transactions in σ that wrote to x before the current transaction read it. All such transactions must have been serialized earlier in $os(\sigma)$, since the serialization ordering follows the conflict graph. The writes of each of these transactions are the same as in σ by the inductive hypothesis, and they were serialized in the same order as in σ because the conflict graph keeps track of write-write conflicts. It follows that the nth transaction in $os(\sigma)$ also reads the same values as it did in σ. Since it also receives the same entangled query answers, the determinism assumption can be applied to infer that it makes the same writes as it did in σ. □

4.5 Execution Model

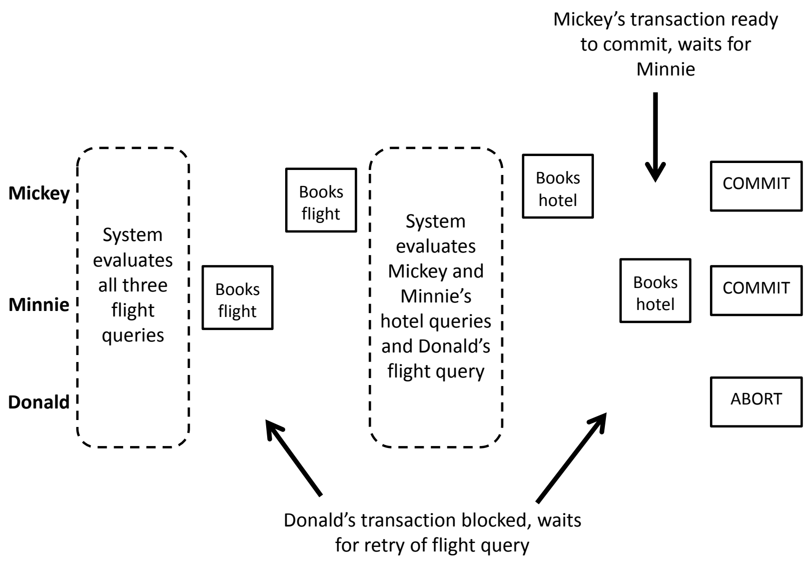

Mickey's transaction ready to commit, waits for Minnie
Mickey
Minnie
Donald
System evaluates all three flight queries
Books flight
Books flight
System evaluates Mickey and Minnie's hotel queries and Donald's flight query
Books hotel
Books hotel
COMMIT
COMMIT
ABORT
Donald's transaction blocked, waits for retry of flight query

Figure 4.4: Example run of three transactions

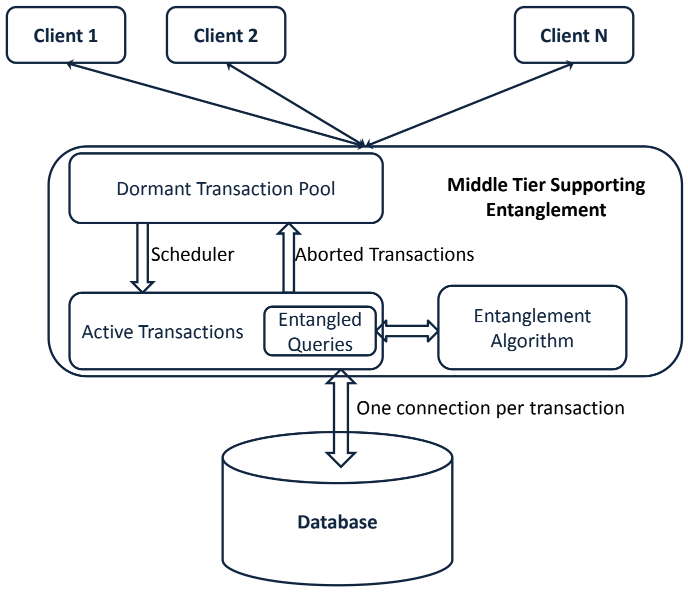


Figure 4.5: System architecture

With the semantic model for entangled transactions in place, we turn to the challenges and performance tradeoffs associated with *execution* models. There is no single best execution model for entangled transactions; the final choice depends on the requirements of the application. In this section, we highlight the tradeoffs and propose a solution suited to realistic scenarios like travel planning.

Interactivity. One of the first key characteristics of an application is whether the transactions are *interactive* or *non-interactive*, or both. Interactive transactions are created by users online, statement by statement. Subsequent statements are constructed dynamically, based on the result of earlier operations. An interactive user may be willing to wait a few minutes for his or her entangled query to find partners and return results. If results are not forthcoming, then the user may decide to abort or issue another command. This interactive model is suited, for example, to social games.

However, in other scenarios such as travel planning, users who want to coordinate will most likely not issue their queries simultaneously and wait for answers at the computer. A non-interactive model is a better fit here: users can be expected to issue entire entangled transactions at once and specify an appropriate timeout. If no partner is found before the timeout expires, then the transaction aborts and is removed from the system. In this paper, we present an execution model for non-interactive transactions; exploring the unique issues associated with interactivity is future work.

Concurrency Control Protocol. As discussed in Section 4.3, several different isolation levels may be appropriate for entangled transactions. The choice of level is up to the system designer and depends on the application's consistency requirements. Whatever the isolation level(s) to be used, the execution model must include a suitable protocol to enforce them.

For scenarios such as collaborative travel planning, a high level of isolation is desirable to ensure consistency of the underlying database. Full isolation can be achieved by enforcing group commits and using a standard strict two-phase locking protocol. This protocol has the additional advantage of admitting isolation relaxations, if desired, by altering the length of time locks are held.

Scheduling. The system needs a policy for handling transactions that cannot currently be matched with entanglement partners. The best choice of policy depends on whether the transactions are interactive and on the isolation level desired. The discussion below makes the assumption that we are working with noninteractive transactions and desire full isolation as defined in Section 4.3.3.

A naïve policy where each transaction blocks at each entangled query until it has found a partner is impractical. This blocked transaction may need to hold locks while it waits, unacceptably delaying the progress of other transactions in the system. One solution is to limit the time for which an entangled query blocks. If a partner does not arrive within a limited time frame, the transaction is aborted and restarted.

It is possible to take this idea further and organize the execution of transactions in discrete batches or *runs*. Each run is an execution of a set of transactions chosen by the scheduler. If an entangled transaction arrives in the system while a run is ongoing, it is suspended and added to a pool of dormant transactions. Designing an optimal scheduling policy is nontrivial. A simple policy is to schedule runs with a particular *frequency*, and include in a run all transactions present in the dormant pool. The frequency can be explicitly given as a time interval, or it can depend on the arrival rate of new transactions. For example, the system may schedule a new run once ten new transactions have arrived.

When a transaction is scheduled in a run, all classical queries and updates that precede the first entangled query are executed. At this point, the transaction blocks. Eventually, all transactions in the run either block, abort or reach the ready to commit state. Now, the system evaluates all pending entangled queries. If an entangled query receives an answer, the transaction is notified and resumes execution. The run terminates when each transaction has either aborted, reached the ready to commit state, or blocked on an entangled query and is unable to proceed. Transactions that are ready to commit and satisfy the group commit constraints (if applicable) are committed. Blocked transactions are aborted and returned to the dormant pool for execution in subsequent runs.

To illustrate run-based transaction scheduling, we walk through an example execution of three entangled transactions. The first is Mickey's transaction from Figure 4.2, and the second is a symmetric transaction by Minnie who wants to coordinate with Mickey. The third transaction follows the same structure, but it involves Donald who is interested in coordinating with Daffy.

Suppose Mickey's and Donald's transactions arrive in the system first. The scheduler creates the first run that includes these two transactions only. The first piece of code in each transaction is the respective flight booking entangled query. Neither transaction is able to progress; therefore, the system immediately aborts the run and returns both transactions to the dormant transaction pool.

Now, Minnie's transaction arrives in the system and is placed in the pool. The scheduler creates a second run containing all three transactions. The execution of this run is shown in Figure 4.4. Mickey and Minnie's transactions are able to execute their first entangled query; they proceed to their respective flight booking code. Donald's entangled query does not receive an answer, so his transaction blocks. Once Mickey and Minnie complete their flight booking, their transactions reach the hotel entangled queries. These are submitted for evaluation together with Donald's flight query, which has not received an answer yet. Again, Mickey and Minnie receive answers and are able to proceed, while Donald does not. Eventually, Mickey and Minnie both reach a state where they are ready to commit, pending their partner's commit. Donald's transaction, however, is still blocking on the flight query. The system recognizes that no-one can proceed further and takes action. Mickey and Minnie's transactions are allowed to commit, while Donald's is aborted again and returned to the dormant transaction pool.

Persistence and Recovery. Entangled transactions come with atomicity and durability requirements, as outlined in Section 4.3.4. It is therefore necessary to ensure correct crash recovery. Standard algorithms must be suitably modified to handle the additional entanglement-related recovery challenges.

In processing entangled transactions, the system maintains additional state to keep track of the transactions that are currently in the system and awaiting partners. It also may be keeping track of who has entangled with whom in order to enforce group commits. This state must be made persistent to ensure correct crash recovery. Further, the recovery algorithm must be entanglement-aware. For example, if two transactions entangle and only one manages to commit prior to a crash, both must be rolled back during recovery.

4.6 Implementation

In this section, we discuss our prototype implementation for the entangled transaction management component in the Youtopia system and present the results of our experimental evaluation.

4.6.1 Prototype

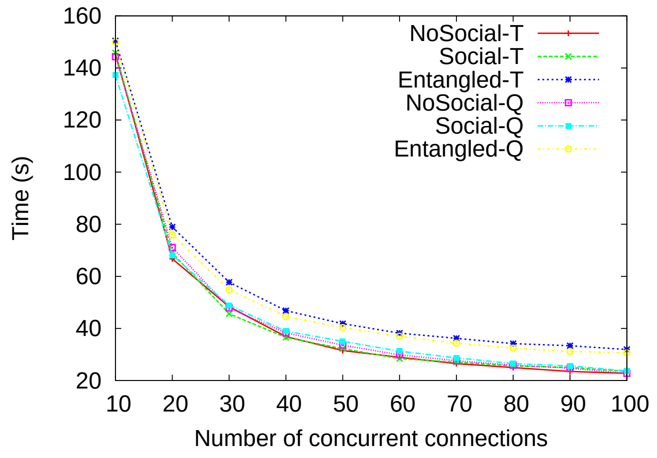
Our prototype implements entangled transaction support in the middle tier, as shown in Figure 4.5. This design makes it easy to port current applications without any significant change to the DBMS or the interface. Alternately, entangled transactions could be implemented within the DBMS itself, which has the potential to improve performance through deep optimizations of the entanglement logic; investigating this alternate architecture is future work.

The prototype is a component within our Youtopia system; it is implemented as a Java application over a MySQL database (version 5.5.0) using the InnoDB engine. It provides an API for clients to manage and query the database, with the added functionality of answering entangled queries and managing entangled transactions. Youtopia users submit transactions (entangled and classical) through a front-end interface. Youtopia executes classical transactions as-is and sends query results back to the client; entangled transactions are handled by our custom component.

The prototype supports the execution model discussed in Section 4.5. It handles non-interactive transactions and uses a run-based scheduling protocol for execution. During runs, entangled queries are evaluated using the algorithm described in [22].

Transactions can be executed at different isolation levels. If full isolation is desired for strong consistency, the system enforces group commits to prevent widowed transactions and uses Strict 2PL to prevent all other isolation anomalies. The locking protocol is implemented using the lock manager of the DBMS.

In our implementation, the middleware is stateless. All relevant system state is serialized and stored in the database to achieve persistency. This allows us to leverage the recovery algorithms implemented in the DBMS to ensure correct



Scalability in concurrent transactions

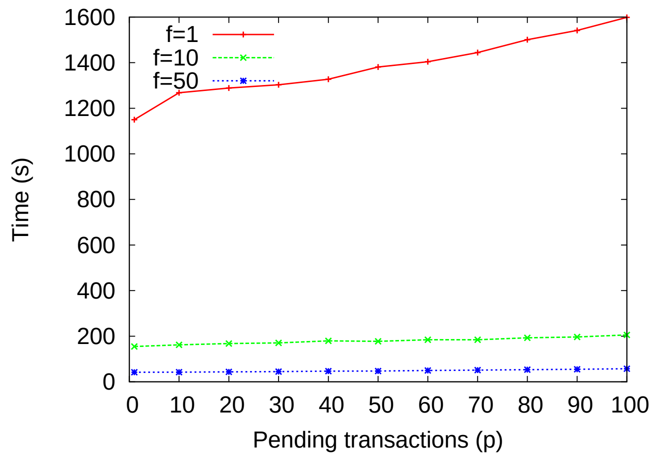


Figure 4.6: Scalability in the number of pending transactions

crash recovery.

4.6.2 Evaluation

In our experiments, we set out to measure the overhead associated with supporting entangled transactions relative to two other abstractions: classical trans-

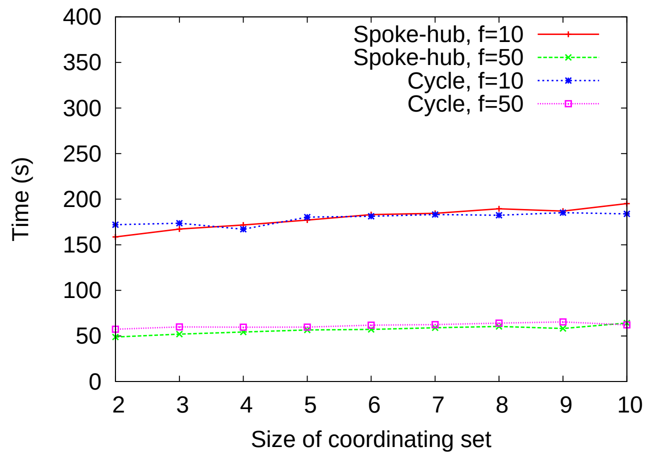


Figure 4.7: Scalability in entangled queries per transaction

actions and non-transactional code containing entangled queries. We also wanted to investigate the tradeoffs associated with the design decisions described in Section 4.5.

Experimental Setup

All our experiments were set in the travel scenario discussed throughout the paper and used a workload of simulated entangled transactions that modeled the output of a front-end social travel application. We created a set of users with friendship relations based on the *Slashdot* social network data [41]. Each transaction was generated by choosing a user and expressing his or her intent to coordinate on flight and/or hotel bookings with a set of friends; for examples, see Section B.1 in the Appendix. Each transaction contained a single entangled query, except where indicated otherwise.

In MySQL, as in most commercial database systems, the amount of concurrency is restricted by the maximum permissible number of connections rather than the computational capacity of the system. This is because only a single transaction may run per connection. We worked within these constraints for the purpose of our experiments. By default, we used 100 concurrent connections, and we examined experimentally the impact of varying that number.

We ran all experiments on a 2.13Ghz Intel Core i7 CPU with 4GB of RAM; the reported values are averages over 3 runs. The standard deviation was less than 2% in each experiment. All experiments involved 10000 (ten thousand) transactions which were run to completion.

Results

Concurrency. In the first experiment, we varied the number of concurrent connections to MySQL from 10 to 100 and investigated the performance of six different workloads. As mentioned, we wanted to compare entangled transactions (`Entangled-T`) to non-entangled transactions in which users make travel bookings based on existing bookings by their friends (`Social-T`). Our third workload, `NoSocial-T`, contained individual travel booking transactions – that is, transactions that made no reference at all to the activity of the user's friends. In addition, for each of the above three workloads, we generated a corresponding non-transactional workload that used the same code without enclosing it within a transaction block. The non-transactional workloads are identified by the suffix `-Q` instead of `-T`. A further discussion of these workloads, together with examples, is found in the Appendix, Section B.1.

For simplicity, all workloads were generated to ensure that all transactions within a single run would be able to coordinate. That is, transactions were submitted in batches designed so that each transaction would find a coordination partner within the same batch.

Figure 4.6.2 shows the results. The time taken to execute any given set of transactions was observed to be inversely proportional to the number of concurrent connections for all three transactional workloads. Although the time taken by `Entangled-T` was always marginally higher compared to `NoSocial-T` (and `Social-T`), the difference was roughly equal to the difference is execution time between `Entangled-Q` and `NoSocial-Q` (and `Social-Q`). This shows that entangled transactions do not impose significant additional overhead relative to classical transactions, except for the extra time needed for the actual evaluation of entangled queries.

Pending Transactions. The first experiment was engineered so that all concurrently submitted entangled transactions would find coordination partners and commit. However, this may not be true in real life. We therefore ran a second experiment where the number of pending transactions remaining at the end of a run, p, was nonzero and varied from 10 to 100. This was achieved by submitting the transactions in carefully designed batches to ensure that each run contained p transactions without coordination partners.

We used three different run scheduling policies with different run frequencies f. We set f in terms of the arrival rate of new transactions in the system and varied it from 1 (start a new run after a single new transaction arrives) to $f = 50$ (start a new run after fifty new transactions arrive).

Figure 4.6 shows the results. As expected, using higher run frequencies had a negative impact on execution time. Moreover, increasing p caused a linear increase in the total execution time. However, this increase was much slower when the run frequency was lower. Clearly, the optimal run frequency for a given workload depends on the expected value of p.

Entanglement Complexity. Our last set of experiments investigated the impact of varying the complexity and structure of the entanglement between transactions. The intuition is that with more complex entanglement structure and more entangled queries per transaction, entanglement may be harder to achieve and transactions may abort more frequently before succeeding.

The specific parameters we varied were the number of entangled queries per transaction and the structure of the coordination. We considered two complex coordination structures. In the `Spoke-hub` structure, a single transaction with multiple entangled queries entangles with a different partner on each query. The `Cyclic` structure is even more complex and involves a cyclic set of entanglement dependencies between a set of entangled transactions.

On all the above workloads, we ran experiments with a run frequency f of 1 and 50. Figure 4.7 gives the results. Increasing the number of entangled queries per transaction increases the total execution time; however, the slope is very small. This suggests that increasing entanglement complexity does not have a significant negative performance impact.

APPENDIX A

ENTANGLED QUERIES

A.1 NP-completness of CNRC

In this section we give a proof of Theorem 3.2.6, i.e. we show that CNRC is NP-complete.

It is clear that the problem is in NP. For hardness, we argue via reduction from 3-SAT.

Assume we are given a formula with k clauses. Construct a graph of the shape indicated in Figure A.1. The graph contains two kinds of nodes: *clause nodes* and *beads*. Intuitively, each clause node corresponds to a clause in the formula, and each set of beads corresponds to a given occurrence of a literal in a clause. The graph contains k clause nodes, and each string of beads is $3k$ nodes long. This gives a total of $9k^2 + k$ nodes.

The graph is colored as follows. First, each clause node is colored with a fresh and distinct color, and none of these k colors are ever reused. Next, the bead strings are colored. We explain how to color a bead string B corresponding to the literal x_i occurring in clause j; the description assumes x_i is unnegated, but the process for negated literals is symmetric.

Some of the beads in the bead string corresponding to x_i may already have assigned colors, so start with the first bead that does not already have a coloring. Identify all occurrences of $\neg x_i$ in clauses numbered $l > j$. For each such occurrence, identify the corresponding bead string B' and choose a fresh color c. Color one bead on B and one bead on B' with c. Discard c, i.e. ensure it never gets used again. When all occurrences of $\neg x_i$ have been processed, color any remaining beads on B with fresh and distinct colors that are never reused.

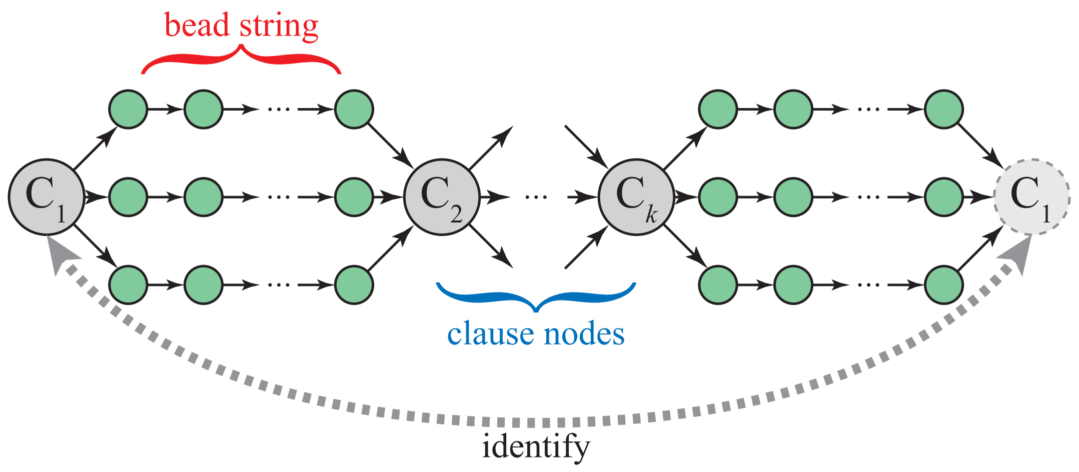


Figure A.1: Graph shape for proof of Theorem 3.2.6

In this graph, a cycle with no repeated colors corresponds to a satisfying assignment to the formula. If we have a cycle with no repeated colors, then we can retrieve a satisfying assignment as follows. Identify the bead strings that are involved in the cycle and the corresponding literals. It is not possible to have both x_i and $\neg x_i$ occurring in this set of literals, as the corresponding bead strings for these literals must share at least one bead color by construction. Thus, the set of literals directly yields a satisfying truth assignment to the formula. Conversely, given a satisfying truth assignment, a cycle with no repeated colors can be obtained by including, for each clause, any bead string that is consistent with the assignment.

APPENDIX B

ENTANGLED TRANSACTIONS

B.1 Examples

In this section we present several examples that we used in our experiments. We created a set of users with friendship relations based on the Slashdot social network data [41].

The schema for our system is as follows:

```
 Reserve(uid, fid)
 Friends(uid1, uid2)
 Flight(source, destination, fid)
 User(uid, hometown)
```

The first workload (No-Social) simulates individual travel booking transactions. It queries the user table to get the source hometown, followed by a query to find flights from this source to the destination. Finally, it makes a reservation for the user.

```
BEGIN TRANSACTION;
SELECT @uid, @hometown FROM User WHERE uid=36513;
SELECT @fid FROM Flight WHERE source=@hometown
                          AND destination='FAT';
INSERT INTO Reserve (uid, fid)
       VALUES (@uid, @fid);
COMMIT;
```

The second workload (Social) also gives a list of friends who live in the same hometown and might be flying. This information is additional to the normal flight reservation.

```
BEGIN TRANSACTION;
SELECT @uid, @hometown FROM User WHERE uid=36513;
SELECT uid2 FROM Friends, User as u1, User as u2
WHERE Friends.uid1=@uid
  AND Friends.uid2=u2.uid
  AND u1.uid=@uid
  AND u1.hometown=u2.hometown
LIMIT 1;
SELECT @fid FROM Flight WHERE source=@hometown
                          AND destination='FAT';
INSERT INTO Reserve (uid, fid)
       VALUES (@uid, @fid);
COMMIT;
```

The third workload (Entangled) checks if a particular friend is also trying to coordinate with the user to make flight reservations.

```
BEGIN TRANSACTION WITH TIMEOUT 2 DAYS;
SELECT @hometown FROM user WHERE uid=45747;
SELECT 36513 AS @uid, 'CAT' AS @destination
INTO ANSWER Reserve
WHERE (36513, 45747) IN
      (SELECT uid1, uid2 FROM
              Friends, User as u1, User as u2
       WHERE Friends.uid1=36513
         AND Friends.uid2=45747
         AND u1.uid=36513
         AND u2.uid=45747
         AND u1.hometown=u2.hometown)
  AND (45747, 'PHF') IN ANSWER Reserve
CHOOSE 1;
SELECT @fid FROM Flight WHERE source=@hometown
                          AND destination=@destination;
INSERT INTO Reserve (uid, fid)
       VALUES (@uid, @fid);
COMMIT;
```

BIBLIOGRAPHY

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

[2] Gustavo Alonso, Divyakant Agrawal, Amr El Abbadi, Mohan Kamath, Roger Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *ICDE*, pages 574–581, 1996.

[3] William Sims Bainbridge. *The Warcraft Civilization*. The MIT Press, 2010.

[4] Richard Bartle. *Designing Virtual Worlds*. New Riders Games, 2003.

[5] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder - a transactional record manager for shared flash. In *CIDR*, pages 9–20, 2011.

[6] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: a distributed architecture for online multiplayer games. In *NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation*, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.

[7] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, New York, NY, USA, 1987. ACM Press.

[8] Chuck Blanchard, Scott Burgess, Young Harvill, Jaron Lanier, Ann Lasko, Mark Oberman, and Mike Teitel. Reality built for two: a virtual reality tool. In *SI3D '90: Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 35–36, New York, NY, USA, 1990. ACM.

[9] James M. Calvin, Alan Dickens, Bob Gaines, Paul Metzger, Dale Miller, and Dan Owen. The simnet virtual world architecture. In *VR*, pages 450–455, 1993.

[10] Tuan Cao, Benjamin Sowell, Marcos Antonio Vaz Salles, Alan J. Demers, and Johannes Gehrke. Brrl: a recovery library for main-memory applications in the cloud. In *SIGMOD Conference*, pages 1233–1236, 2011.

[11] Vincent Conitzer and Tuomas Sandholm. Expressive negotiation over donations to charities. In *ACM Conference on Electronic Commerce*, pages 51–60, 2004.

[12] Sanjay Dalal, Sazi Temel, Mark Little, Mark Potts, and Jim Webber. Coordinating business transactions on the web. *IEEE Internet Computing*, 7(1):30–39, 2003.

[13] Sanjay Dalal, Sazi Temel, Mark Little, Mark Potts, and Jim Webber. Coordinating business transactions on the web. *IEEE Internet Computing*, 7(1):30–39, 2003.

[14] B. Dalton. Online gaming architecture: Dealing with the real-time data crunch in mmos. In *Proceedings of Austin GDC*, 2007.

[15] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[16] Alex Delis and Nick Roussopoulos. Performance and scalability of client-server database architectures. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 610–623, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[17] Thomas A. Funkhouser. RING: A client-server system for multi-user virtual environments. In *Symposium on Interactive 3D Graphics*, pages 85–92, 209, 1995.

[18] Hector Garcia-Molina and Kenneth Salem. Sagas. In Umeshwar Dayal and Irving L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 249–259. ACM Press, 1987.

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

[20] Martin Grohe and Luc Segoufin. When is the evaluation of conjunctive queries tractable. In *In Proceedings of the 33rd ACM Symposium on Theory of Computing*, pages 657–666, 2001.

[21] Nitin Gupta, Alan J. Demers, Johannes Gehrke, Philipp Unterbrunner, and Walker M. White. Scalability for virtual worlds. In *ICDE*, pages 1311–1314, 2009.

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

[23] Nitin Gupta, Milos Nikolic, Sudip Roy, Lucja Kot, Johannes Gehrke, and Christoph Koch. Entangled transactions. *PVLDB*, 3(1), 2011.

[24] Olof Hagsand, Rodger Lea, and M&#229;rten Stenius. Using spatial techniques to decrease message passing in a distributed ve system. In *VRML '97: Proceedings of the second symposium on Virtual reality modeling language*, pages 7–ff., New York, NY, USA, 1997. ACM Press.

[25] Seunghyun Han, Mingyu Lim, and Dongman Lee. Scalable interest management using interest group based filtering for large networked virtual environments. In *VRST '00: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 103–108, New York, NY, USA, 2000. ACM.

[26] IBM Corporation. Web services transactions specification (WS-T). www.ibm.com/developerworks/library/ws-coor/.

[27] Simon L. Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *POPL*, pages 295–308, 1996.

[28] Patric Kabus, Wesley W. Terpstra, Mariano Cilia, and Alejandro P. Buchmann. Addressing cheating in distributed mmogs. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–6, New York, NY, USA, 2005. ACM.

[29] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1:1496–1499, August 2008.

[30] Rick Kazman. Making WAVES: On the design of architectures for low-end distributed virtual environments. In *VR*, pages 443–449, 1993.

[31] T. Keating. Dupes, speed hacks and black holes: How players cheat in MMOs. In *Proc. Austin GDC*, Austin, TX, September 2007.

[32] Richard P. King, Nagui Halim, Hector Garcia-Molina, and Christos A. Polyzois. Management of a remote backup copy for disaster recovery. *ACM Trans. Database Syst.*, 16:338–368, May 1991.

[33] Raphael R. Koster. From instancing to worldy games.

[34] 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.

[35] 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.

[36] G. Koutrika, B. Bercovitz, R. Ikeda, F. Kaliszan, H. Liou, Z. Mohammadi Zadeh, and H. Garcia-Molina. Social systems: Can we do more than just poke friends? In *CIDR*, 2009.

[37] S. Kumar, J. Chhugani, Changkyu Kim, Daehyun Kim, A. Nguyen, P. Dubey, C. Bienia, and Youngmin Kim. Second life and the new generation of virtual worlds. *Computer*, 41(9):46 –53, sept. 2008.

[38] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

[39] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, 2007.

[40] J. Leigh, A. Johnson, and T. DeFanti. Scaling virtual worlds: Simulation requirements and challenges. In *Virtual Reality: Research, Development and Applications*, volume 2, pages 217–237, 1997.

[41] Jure Leskovec and Ana Pavlisic. Slashdot data. http://snap.stanford.edu/data/soc-Slashdot0902.html.

[42] Linden Labs. Second life.

[43] Huaiyu Liu, Mic Bowman, Robert Adams, John Hurliman, and Dan Lake. Scaling virtual worlds: Simulation requirements and challenges. In *Winter Simulation Conference*, pages 778–790, 2010.

[44] Nancy A. Lynch and Michael Merritt. Introduction to the theory of nested transactions. *Theor. Comput. Sci.*, 62(1-2):123, 1988.

[45] David Martin, Johannes Gehrke, and Joseph Halpern. Toward expressive and scalable sponsored search auctions. In *ICDE*, 2008.

[46] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[47] Krithi Ramamritham Mohan Kamath. Failure handling and coordinated execution of concurrent workflows. In *ICDE*, 1998.

[48] Graham Morgan, Fengyun Lu, and Kier Storey. Interest management middleware for networked games. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 57–64, New York, NY, USA, 2005. ACM.

[49] Beatrice Ng, Antonio Si, Rynson W.H. Lau, and Frederick W.B. Li. A multi-server architecture for distributed virtual walkthrough. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 163–170, New York, NY, USA, 2002. ACM.

[50] Marian H. Nodine and Stanley B. Zdonik. Cooperative transaction hierarchies: A transaction model to support design applications. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 83–94. Morgan Kaufmann, 1990.

[51] Ioannis Pantazopoulos and Spyros Tzafestas. Occlusion culling algorithms: A comprehensive survey. *J. Intell. Robotics Syst.*, 35(2):123–156, 2002.

[52] Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. Bayou: replicated database services for world-wide applications. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 275–280, New York, NY, USA, 1996. ACM.

[53] Christos A. Polyzois and Héctor García-Molina. Evaluation of remote backup algorithms for transaction-processing systems. *ACM Trans. Database Syst.*, 19:423–449, September 1994.

[54] Calton Pu, Gail E. Kaiser, and Norman C. Hutchinson. Split-transactions for open-ended activities. In François Bancilhon and David J. DeWitt, editors, *Fourteenth International Conference on Very Large Data Bases, August 29 - September 1, 1988, Los Angeles, California, USA, Proceedings*, pages 26–37. Morgan Kaufmann, 1988.

[55] Python Software Foundation. Stackless python.

[56] John Reppy. *Concurrent Programming in ML*. Cambridge University, 1999.

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

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

[59] Ahmad-Reza Sadeghi and Christian Stüble. Taming ”trusted platforms” by operating system design. In *WISA*, pages 286–302, 2003.

[60] Ahmad-Reza Sadeghi and Christian Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *NSPW '04: Proceedings of the 2004 workshop on New security paradigms*, pages 67–77, New York, NY, USA, 2004. ACM.

[61] Chris Shaw and Mark Green. The MR toolkit peers package and experiment. In *VR*, pages 463–469, 1993.

[62] Laurent Siklóssy and Jean-Louis Laurière. Removing restrictions in the relational data base model: An application of problem solving techniques. In *AAAI*, pages 310–313, 1982.

[63] Sulake Corporation. Habbo hotel.

[64] Hamdy A. Taha. Introduction to simnet v2.0. In *WSC '88: Proceedings of the 20th conference on Winter simulation*, pages 93–101, New York, NY, USA, 1988. ACM.

[65] Alice Taylor. The problem with World of Warcraft.

[66] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *Proc. VLDB Endow.*, 3:70–80, September 2010.

[67] Tobold. Servers and critical mass.

[68] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1996.

[69] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An inte-

grated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[70] Walker White, Alan Demers, Christoph Koch, Johannes Gehrke, and Rajmohan Rajagopalan. Scaling games to epic proportions. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 31–42, New York, NY, USA, 2007. ACM.

[71] Walker M. White, Christoph Koch, Johannes Gehrke, and Alan J. Demers. Better scripts, better games. *ACM Queue*, 6(7):18–25, 2008.

[72] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1995.

[73] Wikia. World of Warcraft universe guide.

[74] Wikipedia. Instance dungeon.