A DISTRIBUTED STORAGE AND QUERY SUBSYSTEM FOR COLLABORATIVE DATA SHARING

Nicholas E. Taylor

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2010

Zachary G. Ives, Associate Professor of Computer and Information Science
Supervisor of Dissertation

Jianbo Shi, Associate Professor of Computer and Information Science
Graduate Group Chairperson

Dissertation Committee

Brian F. Cooper, Principal Research Scientist, Yahoo! Research

Susan B. Davidson, Professor of Computer and Information Science

Andreas Haeberlen, Assistant Professor of Computer and Information Science

Boon Thau Loo, Assistant Professor of Computer and Information Science

A Distributed Storage and Query Subsystem for Collaborative Data Sharing

COPYRIGHT

Nicholas E. Taylor

2010

Acknowledgements

The work described in this dissertation would not have been possible without the assistance, guidance, advice, and support of many people. In particular, I would like to extend my very heartfelt thanks to my advisor, Zack Ives. You were always helpful and accessible, and gave me the freedom to do what I wanted to do with only a few gentle nudges in the right direction. I really appreciate all the work you did on my behalf.

I would like to thank my thesis committee, Professors Susan Davidson, Andreas Haeberlen, and Boon Thau Loo of the Computer and Information Science Department, and Brian Cooper from Yahoo! Research, who read and gave helpful comments on several versions of this thesis. Your service is very much appreciated. I would also like to thank my WPE-II committee members, Susan Davidson, Boon Thau Loo, and Jonathan Smith, for their efforts and assistance.

I would also express my thanks to the many other wonderful professors at Penn that have taught me and provided guidance and mentoring. In particular, I would like to recognize Val Tannen, Susan Davidson, Amir Roth, Boon Thau Loo, Sampath Kannan, and Zack Ives.

I also really appreciate the constructive criticism, proofreading, technical assistance, and camaraderie of my fellow doctoral students at Penn. I especially appreciate the GRW-561 gang, with whom I spent countless hours over the past six years: TJ Green, Greg Karvounarakis, Nate Foster, and Svilen Mihaylov. I'd also like to extend a special thanks to the other students of the Database Group, in particular Marie Jacobs and Mengmeng Liu, and to my PhD classmate chums, namely Jenn Wortman Vaughan, Jeff Vaughan, Drew Hilton, Aaron Bohannon, Kuzman Ganchev, and Karl Mazurak.

I'd like to thank to the support staff that have made my life at Penn much easier. In particular, I'd like to thank Mike Felker, the CIS graduate coordinator; Cheryl Hickey, our administrative coordinator; Lillian Thomas and Mark West from the Moore Business Office; and our Database Group programmers, Sam Donnelly, John Frommeyer, and Olivier Biton.

I'd like to thank my parents for their support, and for showing me that science (albeit the more wet kind) is both interesting and fun.

I'd like to thank the many friends that I've made during my grad school career. You guys, especially Jeff, Jenn, Dave, Rob, Drew, Kristin, Colleen, Michelle, and Heather, made Philly a fun place to call home for the past six years.

Last, but certainly not the least, I'd like to thank Dan for putting up with me, especially when I've been grouchy and stressed out about this thesis. You've really kept me going though this past year, and I can't tell you how much your support has meant to me.

This work was supported by NSF grants IIS-0477972, IIS-0713267, and IIS-0513778, and DARPA grants HRO001-0-1-0016 and HRO1107-1-0029. Amazon.com, Inc. kindly supplied us free use of their EC2 cloud computing service.

ABSTRACT

A Distributed Storage and Query Subsystem for Collaborative Data Sharing

Nicholas E. Taylor

Supervisor: Zachary G. Ives

Cooperative management of data is a difficult challenge. In the absence of a central authority, there is often no single data format, and users may not even agree on what is true and what is not. The data is typically not static and will evolve over time, leading to issues of staleness and conflicting changes. Dedicated machines to run a management system may not be available, and furthermore the machines supplied by the users to run the system may be unreliable or only transiently available. A reliable system must be built over these machines, and should be self-configuring and self-tuning, to avoid placing an undue burden on end users that are unwilling or unable to manage it themselves.

The ORCHESTRA collaborative data sharing system responds to these challenges by providing a general approach for propagating updates between a heterogeneous collection of peer databases, which are connected by high-level rules that specify the correspondences between them. The system maintains these correspondences while enforcing trust conditions to filter the data from other databases, maintaining transactional atomicity, and respecting database integrity constraints. In this thesis, I detail my work on the semantics of transactional atomicity and dependency in this context, which lead to a general reconciliation algorithm; I also describe the prototype centralized and peer-to-peer implementations of ORCHESTRA. I then develop a specialized reliable peer-to-peer storage and query processor that will enable the logging and computation needed to maintain an ORCHESTRA instance to be distributed. I show ways to extend this system to recover from node failure, to perform load balancing to ensure even distribution of work, and to compensate for node heterogeneity and data skew.

Contents

1 Introduction **1**

1.1 Motivation . . . 3

1.2 Outline . . . 5

2 The Collaborative Data Sharing Model **8**

2.1 Participants, Schemas, and Mappings . . . 9

2.2 System operation . . . 11

2.3 Cloud Substrate for ORCHESTRA . . . 22

3 Reconciliation: Algorithms and Implementation **25**

3.1 CDSS Properties . . . 26

3.2 Reconciliation . . . 34

3.3 Reconciliation Algorithms . . . 45

3.4 Update Store . . . 52

3.5 Experimental Analysis . . . 59

3.6 Conclusions and Analysis . . . 67

4 Reliable, Declarative Peer-to-Peer Storage and Computation **68**

4.1 System Architecture and Requirements . . . 71

4.2 Hashing-Based Substrate . . . 75

4.3 Versioned Data Storage . . . 79

4.4 Reliable Query Execution . . . 87

4.5 Experimental Evaluation . . . 98

4.6 Conclusions and Analysis . . . 119

5 Load Balancing **120**
5.1 Partitioning Flexibility 123
5.2 Partitioning as Constrained Optimization 127
5.3 Experimental Analysis 135
5.4 Conclusions 162

6 Related Work **163**
6.1 Data Transformation and Integration 163
6.2 Consistency and Data Sharing 171
6.3 Distributed Query Processing 180
6.4 Load Balancing in Distributed Hash Tables 188

7 Future Work **192**
7.1 The CDSS Model 192
7.2 Distributed Storage and Querying 193
7.3 Load Balancing 195

8 Conclusions **197**

Bibliography **199**

List of Figures

1.1 Logical Model of a CDSS 4
1.2 Physical Model of a CDSS 5

2.1 Example CDSS Instance 10
2.2 Basic architecture of Orchestra 13
2.3 Provenance Graph 20
2.4 Internal CDSS representation of a relation 22

3.1 Sample bioinformatics CDSS 26
3.2 Example Reconciliation 33
3.3 Example transaction dependency graph 38
3.4 Comparison of reconciliation algorithms and update stores 46
3.5 Epoch publication in the DHT store 56
3.6 Example transaction retrieval in the DHT store 57
3.7 Effect of number of participants on execution time 62
3.8 Effect on execution time of reconciliation interval 63
3.9 Effect of number of participants on state ratio 65
3.10 The effect on state ratio of varying reconciliation interval 65
3.11 Effect of transaction size on state ration 66

4.1 Basic architecture of Orchestra 71
4.2 Schemes for partitioning the DHT key space among the participants 75
4.3 Storage scheme 83
4.4 Example versioned relation state 85

4.5 Final versioned relation structures 86
4.6 Example versioned relation lookup 86
4.7 Distributed query plan for Example 3. 91
4.8 Distributed query plan for Example 4. 93
4.9 Running time: STBenchmark, 800K tuples/relation, 1-16 nodes 101
4.10 Running time: TPC-H Scale Factor 0.5, 1-16 nodes 101
4.11 Normalized running time: STBenchmark, 800K tuples/relation, 1-16 nodes 102
4.12 Normalized running time: TPC-H Scale Factor 0.5, 1-16 node 102
4.13 Network traffic: STBenchmark, 800K tuples/relation, 1-16 nodes 104
4.14 Network traffic: TPC-H Scale Factor 0.5, 1-16 nodes 104
4.15 Per-node network traffic: STBenchmark, 800K tuples/relation, 1-16 nodes 105
4.16 Per-node network traffic: TPC-H scale factor 0.5, 1-16 nodes 105
4.17 Running time vs. data size, STBenchmark, 8 nodes 106
4.18 Running time vs. data size, TPC-H, 8 nodes 106
4.19 Network traffic vs. data size, STBenchmark, 8 nodes 107
4.20 Network traffic vs. data size, TPC-H, 8 nodes 107
4.21 Running time vs. per-node bandwidth, 8 nodes, TPC-H scale factor 4 108
4.22 Running time vs. added network latency, 16 nodes, TPC-H scale factor 1 109
4.23 Running time vs. number of higher-latency nodes, 16 nodes, TPC-H scale factor 1 . . . 110
4.24 Running time vs. latency variability, 16 nodes, TPC-H scale factor 1 110
4.25 Running time vs. number of slow nodes, 16 nodes, TPC-H scale factor 1 111
4.26 Total traffic on EC2 nodes, TPC-H scale factor 10 113
4.27 Per-node traffic on EC2 nodes, TPC-H scale factor 10 113
4.28 Execution time on EC2 nodes, TPC-H scale factor 10 114
4.29 Normalized execution time on EC2 nodes, TPC-H scale factor 10 114
4.30 Total traffic on EC2 and cluster nodes, TPC-H scale factor 2 115
4.31 Per-node traffic on EC2 and cluster nodes, TPC-H scale factor 2 115
4.32 Execution time on EC2 and cluster nodes, TPC-H scale factor 2 116
4.33 Performance of incremental recovery 117
4.34 Execution time overhead of incremental computation 118
4.35 Network traffic overhead of incremental recovery 118

5.1 Partitioning of the DHT key space 123
5.2 Optimized partitioning of the DHT key space 125
5.3 Example Minion input file 133
5.4 Initial quality of partitionings. 137
5.5 Initial quality of partitionings as system size changes 138
5.6 Effect of optimization time on balancing quality 139
5.7 Effect of replication factor on balancing quality 141
5.8 Effect of number of nodes on balancing quality 142
5.9 Effect of number of nodes on optimization time 143
5.10 Replication and more nodes 144
5.11 Effect of number of chunks on balancing quality 145
5.12 Relations for BioJoin benchmark 148
5.13 Effect of balancing on TPC-H performance 150
5.14 Effect of balancing on BioJoin performance 151
5.15 Effects of node heterogeneity on performance of TPC-H queries 153
5.16 Effects of node heterogeneity on performance of star schema queries 154
5.17 Effects of node heterogeneity on performance of BioJoin query 155
5.18 Effects of data skew on performance of TPC-H queries 157
5.19 Effects of data skew on performance of star schema queries 158
5.20 Performance comparison with Derby, large server 159
5.21 Performance comparison with Derby, proportional data 160
5.22 Performance comparison with commercial RDBMS 161

6.1 Virtual data integration 164
6.2 Data warehousing 165
6.3 Schema mappings in Piazza 170
6.4 Branching in a version control system 173
6.5 Virtual servers in CFS 188

List of Algorithms

3.1 ReconcileUpdates procedure 48

3.2 CheckState helper function 49

3.3 FindConflicts helper function 50

3.4 DoGroup helper function 50

3.5 UpdateConflicts helper function 51

4.1 DistributedRetrieve procedure 87

Chapter 1

Introduction

With the advent of the Internet, it is much easier than ever before for people to share information. Unfortunately, the mere availability of data doesn't mean that it is directly usable. Variations in how the information is formatted, the representation used, how people interpret it, and even opinions about what is true and what is false make it difficult to share information between different sites, organizations, or companies. In full generality, translating data is a very difficult problem. Highly structured information, however, such as that stored in a database, typically has a consistent semantic interpretation; this is necessary in order to pose complex queries over it. Translation between databases can therefore often be done in an automatic or at least semi-automatic fashion.

Data integration, as the process of combining information from different databases is known, is a rich field that has been the focus of much work by the database community over the past several decades. Various approaches have been taken, exploring both the theoretical limits of data integration, and the complexities of implementing data integration systems. In this thesis, I describe my contributions to ORCHESTRA, a Collaborative Data Sharing System (CDSS), which has been under development at the University of Pennsylvania since 2003.

While ORCHESTRA builds upon and is inspired by previous data integration work, its focus is somewhat different. ORCHESTRA places a strong emphasis on data consistency, data provenance, and ease of use. An instance can grow organically and scale gracefully as new users join it, both in terms of the logical specification of the relationships between databases and the actual system that implements that logical specification. ORCHESTRA implements a data sharing model in which participants have a local database instance which they periodically "synchronize" with a cloud-style service.

The cloud service has two main tasks: to permanently archive all versions of the participants' database instances, and to enable exchange of data between participants' instances, resolving any conflicts that may arise and translating data as needed between database schemas. The service is self-configuring and highly reliable through a peer-to-peer architecture emphasizing automatic failover, data replication, and good processing performance.

In order to implement this vision of a CDSS, ORCHESTRA needs a variety of interacting components. A key challenge in building ORCHESTRA is to enable to avoid the need for dedicated servers, maintenance, and support staff while ensuring high availability. We therefore implement all of ORCHESTRA over a "cloud" of low-powered nodes provided by the participants. Individual nodes in this cloud may fail or become temporarily unavailable, but ORCHESTRA should continue to enable data sharing among all of its users. Therefore, all of the following components must execute over this cloud of nodes:

- an *update exchange* engine, that generates derives queries to translate updates between databases from the high-level mappings that connect them,
- a *versioned storage system* that archives and disseminates data and updates to data,
- a *query system* that executes the queries generated by the update exchange engine over the versioned storage, and
- a *dependency and conflict checking system* that enforces data consistency.

In this thesis, I discuss the design and implementation of these ORCHESTRA components, with particular emphasis on the distribution of work over a collection of unreliable, possibly geographically dispersed, nodes.

Being a large project project, ORCHESTRA was joint work with a number of fellow graduate students and faculty members. In particular, the design of the update exchange engine (and the work on the provenance of data and updates upon which it depends) was the work of others. I will review this work as background for mine. The majority of the thesis will focus on the remaining components. I will describe the implementation of a reliable peer-to-peer storage and querying system, and show that it offers high performance and resilience to failure. I will also discuss ORCHESTRA's consistency model, and how we use dependency tracking and integrity constraints to maintain a high level of data consistency.

1.1 Motivation

The problem of sharing structured data among multiple sites has a long history, and it has been visited in many different settings. There is, however, a general assumption that the end goal is to create a single data instance across all sources, and to perhaps provide a different view for each user. The main focus of such work is therefore on translating data between the different sources to achieve this. Data consistency is either ignored, or performed after-the-fact through data cleaning or data "repair" schemes, which can end up making somewhat arbitrary choices between possible alternatives.

Collaborative data sharing begins with the general observation that a single data instance representing global truth is, in many cases, neither possible nor desirable. Truth, like beauty, can be in the eye of the beholder, or at least dependent on what one's precise interpretation of the meaning of a data item is, and beliefs about other related data. The solution to the needs of collaborating users is therefore not to provide one globally consistent data instance, but rather to give each participant a *custom*, internally consistent instance with the data that this particular participant accepts as authoritative, translated into its native format. This instance should contain data "overlapping" with that at other sites, as well as certain data items that diverge from the others. Translating data between data instances is still an important component of collaborative data sharing, but it must be done in a way that allows divergence and incomplete translation to respect users' beliefs, preferences, and interests.

A major challenge in supporting this data sharing model lies in the fact that every database instance is subject to updates, in the form of transactions. The assignment of multiple updates to a single transaction indicates that these updates are in some way related, and should either be applied together or not at all (i.e. they are *atomic transactions*, the term typically used in the database literature). Not all of these transactions may originate from sources trusted by all parties. Furthermore, some of the transactions may be inapplicable to a data instance, due to integrity constraints or incompatibility with other transactions also under consideration. Hence, each site must decide whether to *accept* an update (in effect, synchronizing a portion of its data instance with the origin of the update), to *reject* it as being untrusted (causing a divergence), or in some cases to *defer* processing an update (if there are multiple conflicting updates with no clear preference). An update-centric model is important, as opposed to simply querying the resulting data values, because it allows sites to reject removals or replacements. However, it adds significant complexity, because one update may depend on a chain of other updates, some of which are trusted and others of which are not. A clean semantics

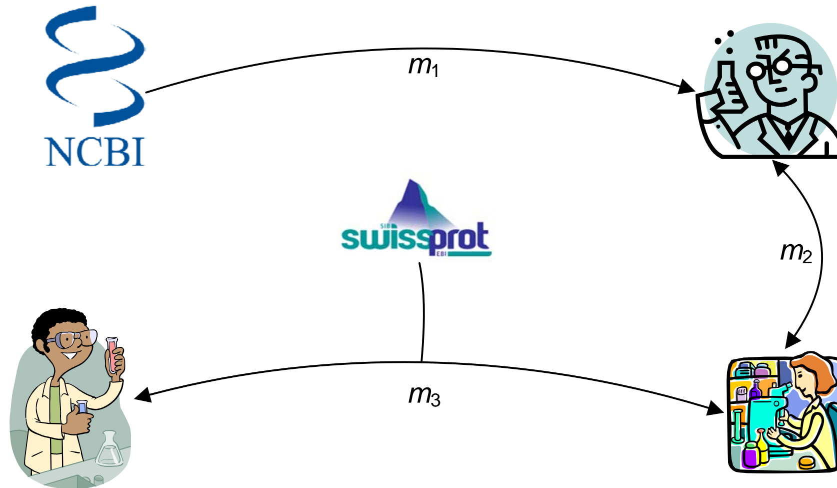


Figure 1.1: Logical conceptual model of a Collaborative Data Sharing System. Data at different sites are related by high-level, declarative mappings. Some sites are only connected indirectly through the composition of multiple mappings.

for update propagation must consider these interactions, as well as different levels of trust, in concert with the problem of translating between differing schemas.

This *loose coupling* of collaborating databases enables new applications that were not supported by prior data integration work. We also wish to reduce the burden of system setup as much as possible, allowing short-lived, *ad hoc* systems. Fixed, global schemas should be avoided. This allows peers to share any of their data, since no limitations are imposed by a global schema, and eases the burden of joining the system. A new participant need just provide a mapping to the database with the most similar schema, instead of to some potentially complex global schema. Additionally, the system implementation should be autonomous and scale seamlessly as databases join and leave the system. There should be no need for a dedicated central server, a database administrator, or really any kind of tuning of any kind.

At the University of Pennsylvania, there has been an ongoing effort both to develop the theoretical underpinnings of collaborative data sharing, and to design and develop Orchestra, an efficient CDSS implementation. Figure 1.1 shows the logical model of a CDSS, wherein multiple independent sites are related by declarative schema mappings. The mappings relate specific sites, and data can flow through a composition of mappings; there is no global unified schema. Figure 1.2 presents a model of the implementation of the Orchestra CDSS. Though the mappings are specified in a peer-to-peer fashion, the CDSS inserts itself as an intermediary between any participant-to-participant communi-

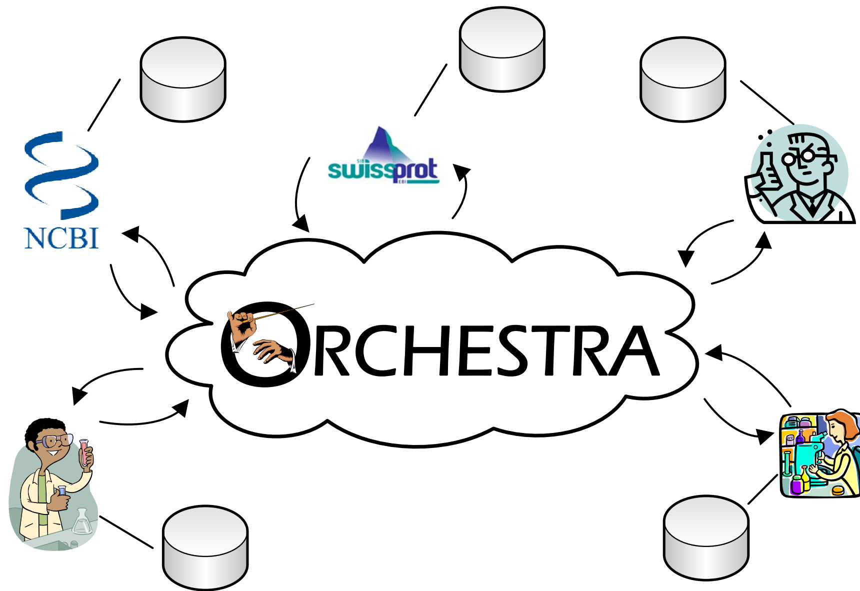


Figure 1.2: Physical implementation of the ORCHESTRA CDSS. Each site maintains its own data instance, which is updated by communicating with the CDSS.

cation. Each participant has a local data instance, which the system maintains by applying translated updates from other participants.

1.2 Outline

As mentioned previously, a system the scale of ORCHESTRA is much too large and complicated for one person to design and implement. Other students and faculty members developed the basic requirements for ORCHESTRA from bioinformatics use cases, worked through the theoretical underpinnings of translating and exchanging updates, and created an initial implementation based on these ideas. While the important steps validated the feasibility of ORCHESTRA, they were largely focused on update translation; they did not consider transactional atomicity or integrity constraints. Additionally, the work was implemented over a single relational database, which was used as storage for both updates and participants' data instances. In this thesis, I show how to add features to ORCHESTRA to address these shortcomings. In particular, I make the following contributions:

- A definition of transactional dependency and mutual consistency between updates in a CDSS,
- An efficient algorithm that chooses a set of consistent updates to apply, while respecting trans-

actional dependencies and user preferences,

- A reliable versioned storage layer, built over a distributed hash table, that is suitable to hold the persistent state needed for a peer-to-peer implementation of ORCHESTRA, and
- A reliable, high-performance peer-to-peer query processing layer, capable of executing the queries needed to translate updates between participant schemas.

This thesis begins with a more detailed motivation of the collaborative data sharing approach. Chapter 2 also gives a detailed introduction to the collaborative data sharing system model, and describes the components of ORCHESTRA that were implemented by others. In particular, it details how the *update exchange* process, introduced in Green et al. (2007a), translates updates in one schema into updates over another schema, and describes how this process enables incremental maintenance of the participant databases in an ORCHESTRA instance.

Chapter 3 describes the development of *reconciliation* in ORCHESTRA, the process by which a participant's local instance is updated with new changes from other participants. I define a notion of dependency *between* transactions, which is necessary to fully respect transactional atomicity. These dependencies are then respected when considering which transactions can be accepted, as are integrity constraints, such as primary keys. I then develop an efficient algorithm to determine a maximal set of mutually compatible, constraint-satisfying transactions, while respecting user preferences. I experimentally validate the performance of this algorithm and explore its effects on various properties of ORCHESTRA instances.

Chapter 4 introduces the second component of this thesis, a peer-to-peer storage and query processing substrate based on the distributed hash table. This substrate allows the storage and data processing needs of ORCHESTRA to be distributed across the participants in the system, instead of using a centralized relational database; additionally, due to its peer-to-peer nature, it is simple to set up and maintain, and can continue to operate in the presence of node failures. Because of the complex queries that can result from mapping composition, good performance is a critical goal. Additionally, for correct ORCHESTRA system operation, the substrate must meet strict reliability requirements for both storage and querying. Taken together, these challenges require a novel approach to peer-to-peer databases, one with an emphasis on reliability and high performance instead of scalability to thousands or millions of nodes. In this thesis, I develop techniques to ensure reliable, consistent access

to mutable data, even in the presence of replication lag and transient node and connectivity failures. These techniques use versioned storage and and a multi-level index structure to verify that the correct version of each data item is used. I explore means to reduce the overhead of creating, updating, and using this index structure. This approach to reliable storage is complemented by techniques for efficient query processing over the reliable data access layer, including support for incremental recomputation of results if a node or connectivity failure occurs during query execution. I experimentally validate query performance using a variety of workloads and under a variety of conditions, including constrained bandwidth and high latency networks. I demonstrate the system's scalability both to large amounts of data and to large numbers of physical nodes. I also show the system's resilience to failure and the benefit of incremental recovery of query results.

Chapter 5 expands on the ideas presented in Chapter 4 to add support for load balancing to improve query performance. It also avoids the need for the custom data distribution scheme using in Chapter 4, which is more susceptible to churn than the partitioning schemes using in much prior peer-to-peer work. I show how the redundant storage of data items in the distributed substrate (necessary for reliability reasons) provides flexibility in routing requests for those data items to particular nodes. By altering the partitioning of data items among nodes, we can achieve good performance without increased susceptibility to churn. We can also alter the partitioning to assign more data to more powerful nodes, and can compensate for data skew by considering the number of data items assigned to each node instead of more abstract routing properties. I show how data partitioning can be posed as a general constrained optimization problem, and solved by a general purpose constraint solver. I also demonstrate the benefit of using optimized partitionings generated in this fashion to compensate for routing imbalance, node heterogeneity, and data skew using several query workloads.

Chapter 6 discusses related work in a variety of areas, including data integration, consistency of distributed data, peer-to-peer systems, and distributed query processing. Chapter 7 offers suggestions for future work. Chapter 8 reviews and summarizes this main contributions of this thesis.

Chapter 2

The Collaborative Data Sharing Model

The collaborative data sharing system (CDSS) was initially proposed in Ives et al. (2005), which described the setting and basic operation, though left as open questions how many of the operations would be performed. Subsequent papers refined and expanded the model, and provided efficient implementations of its operations. The first such paper is Taylor and Ives (2006), which forms the basis of Chapter 3 of this thesis and describes an efficient distributed implementation of Orchestra for the special case where all participants share the same schema. Later work, including Green et al. (2007a), Green et al. (2007c), Ives et al. (2008), Karvounarakis and Ives (2008), Karvounarakis (2009), and Green (2009) explore the theoretical questions that arise when considering translation between schemas, and demonstrate that these can be implemented efficiently in a centralized relational database. Taylor and Ives (2010), which forms the basis for Chapter 4 of this thesis, describes a distributed query processing and data storage system designed to provide a base for a fully functional, distributed Orchestra implementation.

In this chapter, I describe the basic features of the Orchestra CDSS. This chapter is structured as follows:

- Section 2.1 describes the Orchestra model, with an emphasis on how mappings relate data from different participants.
- Section 2.2 begins with a high-level description of the Orchestra system operations.
- Section 2.2 described the *update exchange* process by which updates over one schema are translated into another using the mappings.

- Section 2.2 describes the ORCHESTRA provenance model, which is both used internally and presented to users so they can understand the origin of data in the system.
- Section 2.3 describes how the components of ORCHESTRA described in this chapter interact with work described in Chapters 3, 4, and 5 of this dissertation. This work completes the ORCHESTRA picture, and provides a distributed "cloud"-based substrate for the ORCHESTRA CDSS.

Where the model in Ives et al. (2005) differs from that of later work, I present the more recent CDSS model.

2.1 Participants, Schemas, and Mappings

A CDSS instance consists of a number of autonomous *participants*, sometimes also referred to as *sites* or even *peers* to emphasize their physical isolation or the decentralized nature of the systems. Each participant p from the set of all participants P has its own local schema Σ_p, which is a set of relations. These relations are the participants' *local data instances*, and are stored in a relational database instance I_p by the participant. They contain the data that the participant is interested in querying and also the data that the participant desires to share with the other participants.

The global schema $\Sigma = \bigcup_{p \in P} \Sigma_p$ is the union of all of the peer schemas. Each site is thought of as having a unique schema, even if its relations are named and structured the same as those of other sites. The relations in Σ are related by a collection of mappings $\mathcal{M}$ in global-local-as-view (GLAV) form (Friedman et al., 1999), or equivalently in the form of tuple-generating dependencies (TGDs), as described in Fagin et al. (2005). GLAV mappings are more powerful than the global-as-view and local-as-view approaches used in earlier data integration work, such as Levy et al. (1996) and Garcia-Molina et al. (1997) respectively. Throughout this thesis, we will express GLAV mappings as TGDs using a syntax similar to Datalog (Ullman, 1988, chap. 3). $\mathcal{M}$ can be viewed as a set of constraints on the global system instance. This web of mappings, in which some databases are only indirectly connected, is inspired by and based on the approach used in peer data management research, and described in Halevy et al. (2003) and Kementsietsidis et al. (2003). We discuss all of these related approaches in much more detail in Section 6.1.

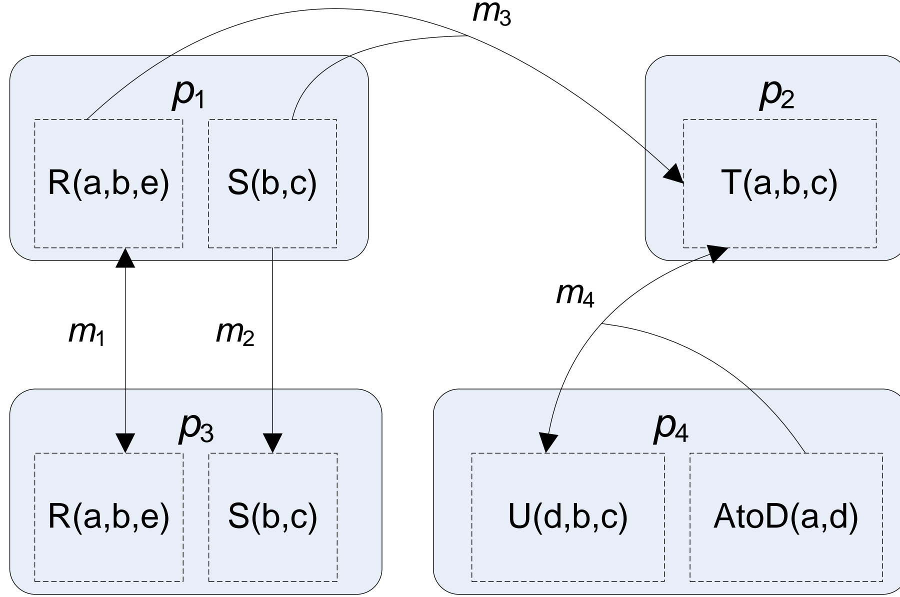


Figure 2.1: Example CDSS Instance

Suppose participants p_1, p_2, p_3, and p_4 are in a CDSS. As shown in Figure 2.1, p_1 and p_3 have the same relations R and S. Mapping m_1 keeps p_1's R and p_3's R in sync, while m_2 copies changes from p_1's S to p_3's. p_3 may make local changes to it's copy of S, but p_1 will never see them. These mappings can be expressed as follows:

$$p_1.R(a, b, e) \leftrightarrow p_3.R(a, b, e) \tag{m_1}$$

$$p_3.S(a, b) \leftarrow p_1.R(a, b) \tag{m_2}$$

More interesting is the relationship between p_1 and p_2. Here, two relations from p_1 are joined to create a single relation in p_2:

$$p_2.T(a, b, c) \leftarrow p_1.R(a, b, e) \wedge p_2.S(b, c) \tag{m_3}$$

Note that, like m_1, this is only a unidirectional mapping. The relationship between p_2 and p_4 also involves a join, but here it is a bidirectional mapping:

$$p_2.T(a, b, c)^* \leftrightarrow p_4.U(d, b, c)^* \wedge p_4.\mathit{AtoD}(a, d) \tag{m_4}$$

In this case, the *AtoD* table translates between the ID a is T and the ID d in U. The use of a such a correspondence table is common in data integration scenarios. The asterisk annotations indicate which

relations may be updated to satisfy the mapping. In this case, the correspondence table will never be updated automatically. More details about bidirectional mappings can be found in Karvounarakis and Ives (2008).

The GLAV mappings used in ORCHESTRA are surprisingly versatile and can express a wide variety of relationships between tables, many of which are demonstrated in this example. They can normalize or denormalize relations, i.e. split a wide relation into multiple relations using a key or reconstruct a wide relation from several normalized ones. They can project away attributes that are needed in one database but not in another. Mapping m_3 in the example combines R and S from p_1 into the wide relation T in p_2, while at the same time projecting away attribute e. They can translate database-specific identifiers using a human-curated correspondence table, as in the example's mapping m_4. With the addition of simple scalar functions, they can perform arithmetic, string operations, and other formatting transformations. These are not shown in the example, since they are not of theoretical interest, but many of the scalar operations supported by traditional database systems nicely complement "pure" Datalog. For example, one could imagine combining two separate attributes for a person's first name and surname into a single field using string concatenation, or performing the inverse by splitting on whitespace. In a scientific setting, one could imagine converting from Ångströms to nanometers by dividing by ten. In short, GLAV mappings are surprisingly versatile, and their use in ORCHESTRA builds upon their successful use in the aforementioned peer data management systems.

2.2 System operation

In general, ORCHESTRA operation is as disconnected as possible, for several reasons. Based on the operational preferences of collaborators in the biological sciences, it was imperative that queries be kept private, to avoid being "scooped." For example, if the set of genes a user references in queries were made public, and it were well-known that the user is involved in cancer research, it might hint that the user believed those genes to play a role in cancer development. To avoid this problem, each participant p maintains its own private instance I_p, which is stored locally, and over which all queries are posed; in this way no information about the queries ever leaves the database holding I_p. The CDSS is responsible for integrating data from other participants into I_p. Additionally, of course, this means that querying can happen even in the event of network disconnection, and that existing

database systems capable of executing complex queries efficiently can be used. Each participant periodically imports data from the rest of the system by performing a *import* operation to bring in newly published trusted updates from other participants and apply them to I_p.

Each participant p also makes changes to I_p, which are grouped into atomic transactions. Initially these changes are stored to a local update log, and can be shared with the rest of the CDSS when the participant chooses to perform a *publish* operation. We chose a batched publish operation (instead of publishing all transactions as they are performed) for several reasons. First, in some scenarios, such as the bioinformatics example of Ives et al. (2005), it is desirable to publish data only after some other event, such as a paper publication or a patent application; by using a batched publish operation we enable this. Second, it allows updates to be performed during a transient network failure; only during the publish operation and the aforementioned import operation does a participant need to be connected to the rest of the system.

Orchestra takes the disconnected mode of operation one step further. In addition to not requiring that any given peer have access to the network when querying its data instance, it also requires that the synchronization operations not depend on any particular other participant being available and connected to the network. If there are many participants and even a small probability that *each* of them is temporarily unavailable, it becomes very unlikely that *all* of them are available at any given moment. Therefore, we cannot simply rely on a participant to store the authoritative archive of the updates it has published. Also, to ensure a complete record of system state is available, and to enable new participants to join at a later date, all published updates must be maintained permanently, even if the participant that published them has left the system.

Instead of having each participant store the updates it has published, all operations in Orchestra involve communicating with a global *update store*. It must at least provide access to the global *update log*, and may perform additional computation to reduce the processing load of the synchronization operations on a participant. The update store can be implemented in several ways. Clearly, one option is to simply store the update log in a relational database, which is a natural and simple way to store what is essentially relational data. This approach was taken by Green et al. (2007a) and Taylor and Ives (2006) consider it as one of two alternatives, as described in Chapter 3. The latter also considers a distributed, peer-to-peer storage system for the update log.

However the updates are stored, the system is responsible for maintaining a permanent record of them, stored in a way that makes performing the import operations easy. In Orchestra, the system

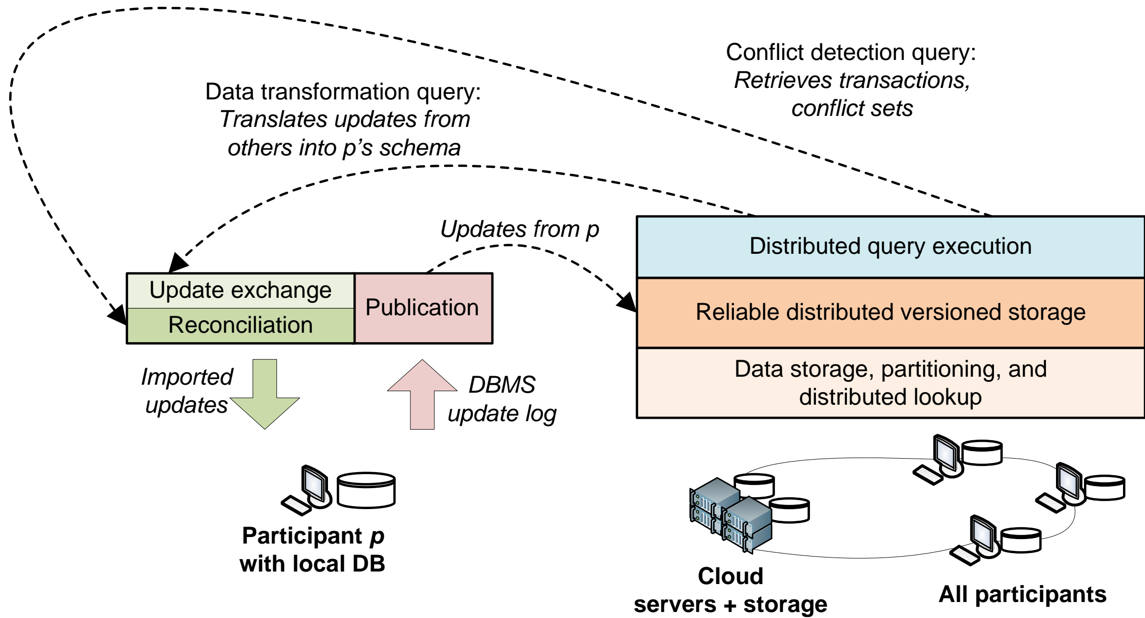


Figure 2.2: Basic architectural components in the ORCHESTRA system, as a participant publishes its update logs and imports data from elsewhere. The boxes on the left show the key components of ORCHESTRA, introduced in Taylor and Ives (2006) and Green et al. (2007a), and described in Chapters 2 and 3. The boxes on the right show how the components of ORCHESTRA interact with and execute over a distributed "cloud" of nodes. This distributed storage and querying engine is the focus of Chapters 4 and 5, which expands on the ideas of Taylor and Ives (2010).

maintains an atomic *epoch* counter, which is incremented for each publish or import operation. Each published group of updates is tagged with its epoch, and the epochs during which each participant performed an import are also recorded. In this way, it is easy to tell which updates are new during a given import operation, as they will have epochs after the preceding one and before the current one.

Figure 2.2 shows ORCHESTRA's architecture, and sketches the dataflow involved in its main operations. Each participant (illustrated on the left) operates a local DBMS with a possibly unique schema, and uses this DBMS to pose queries and make updates. It invokes ORCHESTRA when it has a stable data instance it wishes to "synchronize" with the world: this involves *publishing* updates from the local DBMS log to the CDSS's versioned storage (which contains the update log), and *importing* updates from elsewhere. The implementation of these operations on the right can for the moment be ignored; we will return to these challenges in later portions of this thesis.

The publish operation is typically simple, and simply involves copying data into the system's persistent storage. The fact that publication can *never* fail due to conflicting data is one of the key dis-

tinctions between a CDSS and a traditional version-control system. Importation, on the other hand, is a complex process involving several steps. Briefly, the steps are

1. **Retrieve newly published updates** from the update log. We describe two simple implementations of this in Section 3.4, and devote Chapter 4 to a more general distributed implementation of the update log.

2. **Translate the new updates** into the importing participant's schema. We describe the *update exchange* procedure used to do this in Section 2.2.

3. **Filter updates using trust conditions** to respect the importing participant's restrictions on data it is willing to accept. This at first glance may seem relatively trivial. However, maintaining a record of the derivation or *provenance* of an update translated from multiple input tuples is complicated but necessary to do sophisticated filtering. We will discuss this problem in detail in Section 2.2.

4. **Filter updates for consistency** by choosing a maximal set of updates that satisfies transactional dependencies and respects integrity constraints. This process is known as *reconciliation*. We put off a discussion of reconciliation until Chapter 3, which both defines concretely what the filtering process should consider and gives a concrete algorithm to perform it.

5. **Apply selected updates** to synchronize the participant's local instance with the rest of the system.

In this chapter, as in Green et al. (2007a), we assume that each update is in its own transaction, that updates do not violate integrity constraints, and that updates are insertions or deletions, not modifications of existing tuples. Additionally, each relation is treated as a set. Duplicate insertions of the same tuple do not cause constraint violations; they are instead treated as different supporting evidence for the same tuple. These relaxations ensure that that the filtering done by the reconciliation process would not eliminate any updates, and in fact makes all updates independent of each other and allows them to be applied in any order. In Chapter 3, we will consider how dependencies between updates influence the order in which they can be applied.

Update Translation and Update Exchange

Much of the complexity in translating updates lies in the fact that that correspondences are specified between relations, as in prior data exchange work like Fagin et al. (2005), presenting a kind of logical mismatch. Applying updates from one participant to another's data therefore entails translating those updates in order to maintain satisfaction of the mappings between them. Let us consider again the scenario shown in Figure 2.1. Since the mappings between p_1 and p_3 are identity mappings, translating an update over $p_1.R$ to one over $p_3.R$ is trivial. More complex is the question of translating updates from p_1 to p_2. Orchestra adopts the *delta rules* approach of Gupta et al. (1993). Each relation is modeled as three relations, its current contents, say R, additions to it, R^+, and deletions from it R^-. Delta rules creates new mappings relating the base, insertion, and deletion relations in the source of a mapping to those relations in the target. For example, mapping m_3 would become

$$T^+(a,b,c) \leftarrow R^+(a,b,e) \wedge S(b,c) \wedge \neg S^-(b,c)$$
$$T^+(a,b,c) \leftarrow R(a,b,e) \wedge S^+(b,c) \wedge \neg R^-(a,b,e)$$
$$T^+(a,b,c) \leftarrow R^+(a,b,e) \wedge S^+(b,c)$$
$$T^-(a,b,c) \leftarrow R^-(a,b,e) \wedge S(b,c)$$
$$T^-(a,b,c) \leftarrow R(a,b,e) \wedge S^-(b,c)$$

The updates from T^+ and T^- can then be applied to T. The closure of the mappings can also be used to translate updates. For example, updates from p_1 may be translated and applied to p_4, without p_2 having to accept them. In this case, the delta rules for the composition of m_3 and m_4 is even more complex:

$$T^+(a,b,c) \leftarrow R^+(a,b,e) \wedge S^+(b,c) \wedge AtoD^+(a,d)$$
$$T^+(a,b,c) \leftarrow R^+(a,b,e) \wedge S^+(b,c) \wedge AtoD(a,d) \wedge \neg AtoD^-(a,d)$$
$$T^+(a,b,c) \leftarrow R^+(a,b,e) \wedge S(b,c) \wedge AtoD^+(a,d) \wedge \neg S^-(b,c)$$
$$T^+(a,b,c) \leftarrow R(a,b,e) \wedge S^+(b,c) \wedge AtoD^+(a,d) \wedge \neg R^-(a,b,e)$$
$$T^+(a,b,c) \leftarrow R(a,b,e) \wedge S(b,c) \wedge AtoD^+(a,d) \wedge \neg R^-(a,b) \wedge \neg S^-(b,c)$$
$$T^+(a,b,c) \leftarrow R(a,b,e) \wedge S^+(b,c) \wedge AtoD(a,d) \wedge \neg R^-(a,b) \wedge \neg AtoD^-(a,d)$$
$$T^+(a,b,c) \leftarrow R^+(a,b,e) \wedge S(b,c) \wedge AtoD(a,d) \wedge \neg S^-(b,c) \wedge \neg AtoD^-(a,d)$$
$$T^-(a,b,c) \leftarrow R^-(a,b,e) \wedge S(b,c) \wedge AtoD(a,d)$$
$$T^-(a,b,c) \leftarrow R^-(a,b,e) \wedge S(b,c) \wedge AtoD(a,d)$$
$$T^-(a,b,c) \leftarrow R^-(a,b,e) \wedge S(b,c) \wedge AtoD(a,d)$$

Clearly, even relatively simple mappings, as in this example, can generate a large number of update translation rules. This makes quick, efficient evaluation of these rules an important feature of any practical CDSS implementation. We will return this challenge in Chapter 4.

An example showing these rules in operation may help to make them more understandable. Let us consider, for simplicity, only peers p_1 and p_2. Suppose the instances were initially empty, and then p_1 inserts some tuples, giving the following delta relations:

R^+

'x'	10	7.6
'x'	10	9.4
'y'	20	8.3
'z'	30	2.9

R^- (empty)

S^+

10	'A'
20	'B'
20	'C'

S^- (empty)

The delta rules approach would then create the following derived delta relations:

T^+

'x'	10	'A'
'y'	20	'B'
'y'	20	'C'

T^- (empty)

Applying these deltas would result in the following instance for p_1 and p_2:

R

'x'	10	7.6
'x'	10	9.4
'y'	20	8.3
'z'	30	2.9

S

10	'A'
20	'B'
20	'C'

T

'x'	10	'A'
'y'	20	'B'
'y'	20	'C'

Suppose then that p_1 performed a modification S(20, 'B') $\rightarrow$ S(20, 'BB'), and deleted R('x', 10, 7.6) and S(20, 'B'), causing the following delta relations:

R^+ (empty)

R^-

'x'	10	7.6

S^+

20	'BB'

S^-

10	'A'
20	'B'

The delta rules would then create the following delta relations for T:

T^+

'y'	10	'BB'

T^-

'x'	10	'A'
'y'	20	'B'

Applying R^-, S^-, and T^- would give the following combined instance:

R

'x'	10	9.4
'y'	20	8.3
'z'	30	2.9

S

10	'A'
20	'BB'
20	'C'

T

'y'	20	'BB'
'y'	20	'C'

This instance is mostly correct, but the tuple R('x', 10, 'A') should be in T, and is not. This is because it had derivations that used two different tuples from R, and the deletion inadvertently canceled out both of them. To address this problem, the original work in Gupta et al. (1993) proposed two different algorithms, one based on counting the number of derivations for each tuple, and another known as DRed that compensates for this over-deletion by trying to rederive any tuples it deleted. The counting algorithm cannot be used in this case, as there may be cycles in the mappings, leading to infinite counts. DRed can be used, and is considered in Green et al. (2007a).

However, to better understand the complete derivation of all data in the system, each tuple is tagged with a record of which tuples it was derived from and how they were used in the mappings. This *provenance* information must be maintained as the updates are propagated through the system. In the next section, we describe the Orchestra provenance model. This provenance model is rich enough to allow for mode sophisticated incremental maintenance approach to handle deletions of tuples. Essentially, the provenance is incrementally maintained, and when there are no longer any complete derivations of a tuple, it is deleted. We describe this in detail in the next section, after a discussion of the provenance model.

Provenance Maintenance and Trust Conditions

The Orchestra provenance model is based on the *provenance semirings* described in Green et al. (2007b). Provenance semirings are very powerful, and generalize the provenance models of Buneman et al. (2001), Cui (2001), and Widom (2005). Here we give a brief example of the provenance

of various tuples in an instance of our running example. The schema and mappings for the example are shown in Figure 2.1. Let us then consider an example where, as in the previous section, p_1 inserts the following data into p_1.R and p_1.S:

p_1.R

'x'	10	7.6	t_1
'x'	10	9.4	t_2
'y'	20	8.3	t_3
'z'	30	2.9	t_4

p_1.S

10	'A'	t_5
20	'B'	t_6
20	'C'	t_7

p_4 also inserts the following data:

p_4.U

'N'	20	'C'	t_8
'O'	35	'D'	t_9

p_4.AtoD

'q'	'O'	t_{10}
'x'	'L'	t_{11}
'z'	'M'	t_{12}

No other peer inserts any data. Note that these "base" tuples have been tagged with identifiers t_1 to t_{12}. After the updates are translated and applied to the other peers, the resulting system instance is as follows:

p_1.R

'x'	10	7.6	$\mathbf{z_1} = t_1 + m_1(\mathbf{z_{16}})$
'x'	10	9.4	$\mathbf{z_2} = t_2 + m_1(\mathbf{z_{17}})$
'y'	20	8.3	$\mathbf{z_3} = t_3 + m_1(\mathbf{z_{18}})$
'z'	30	2.9	$\mathbf{z_4} = t_4 + m_1(\mathbf{z_{19}})$

p_1.S

10	'A'	$\mathbf{z_5} = t_5$
20	'B'	$\mathbf{z_6} = t_6$
20	'C'	$\mathbf{z_7} = t_7$

p_2.T

'q'	35	'D'	$\mathbf{z_{13}} = m_4(\mathbf{z_9} \cdot \mathbf{z_{10}})$
'x'	10	'A'	$\mathbf{z_{14}} = m_3(\mathbf{z_1} \cdot \mathbf{z_5}) + m_3(\mathbf{z_2} \cdot \mathbf{z_5}) + m_4(\mathbf{z_{24}} \cdot \mathbf{z_{11}})$
'y'	20	'B'	$\mathbf{z_{15}} = m_3(\mathbf{z_3} \cdot \mathbf{z_6})$
'y'	20	'C'	$\mathbf{z_{16}} = m_3(\mathbf{z_3} \cdot \mathbf{z_7})$

p3.R

'x'	10	7.6	$\mathbf{z_{17}} = m_1(\mathbf{z_1})$
'x'	10	9.4	$\mathbf{z_{18}} = m_1(\mathbf{z_2})$
'y'	20	8.3	$\mathbf{z_{19}} = m_1(\mathbf{z_3})$
'z'	30	2.9	$\mathbf{z_{20}} = m_1(\mathbf{z_4})$

p3.S

10	'A'	$\mathbf{z_{21}} = m_2(\mathbf{z_5})$
20	'B'	$\mathbf{z_{22}} = m_2(\mathbf{z_6})$
20	'C'	$\mathbf{z_{23}} = m_2(\mathbf{z_7})$

p4.U

'L'	10	'A'	$\mathbf{z_{24}} = m_4(\mathbf{z_{14}} \cdot \mathbf{z_{10}})$
'N'	20	'C'	$\mathbf{z_8} = t_8$
'O'	35	'D'	$\mathbf{z_9} = t_9 + m_4(\mathbf{z_{13}} \cdot \mathbf{z_{10}})$

p4.AtoD

'q'	'O'	$\mathbf{z_{10}} = t_{10}$
'x'	'L'	$\mathbf{z_{11}} = t_{11}$
'z'	'M'	$\mathbf{z_{12}} = t_{12}$

Observe that each tuple in the output instance is tagged with a provenance $\mathbf{z_i}$. This represents the *single-step provenance* of that tuples, i.e. all of the tuples and mappings from which it can be immediately derived. Addition indicates alternate derivations of the same tuple, and multiplication indicates that a derivation uses multiple tuples. The provenance of base tuples includes their identifiers, in addition to any ways they may be derivable through mappings. For example, the provenance of p_1.R('x', 10, 7.6) indicates that this tuple was manually inserted, but is also derivable though mapping m_1 from a tuple with provenance $\mathbf{z_{16}}$. The tuple p_2.T('y', 20, 'B') is derivable though mapping m_3 using two tuples, one of which has provenance $\mathbf{z_3}$ and the other $\mathbf{z_6}$.

In effect, the provenance expressions $\mathbf{z_1} \ldots \mathbf{z_{24}}$ form a system of equations. In a simpler case (i.e. if the mappings were not bidirectional, or there are no cycles in the mappings), there would be a finite solution to this set of equations. Indeed, there is for certain tuples. For example, $\mathbf{z_{20}}$ has finite value $m_2(t_6)$. However, for most tuples in the system, cycles in the mappings cause there to be no finite solution. The use of a system of equations allows a finite representation of this infinite solution.

One can also think of each of these single-step provenance expressions as adding nodes and edges to what Green et al. (2007a) refers to as the *provenance graph*. This directed graph has tuple nodes, which represent values in the relations, mappings nodes, which connect tuple nodes to tuples that can be derived from them, and token nodes, which annotate base tuples with their provenance tokens. A provenance graph for part of our running example can be seen in Figure 2.3.

Participants in a ORCHESTRA instance supply trust conditions stating which mappings, tuples, and peers (or any combination thereof) they trust. In our example, p_2 might distrust any tuple in p_1.R where the third attribute is less than 9; this means that, for p_1, $\mathbf{z_1}$ would be distrusted and therefore

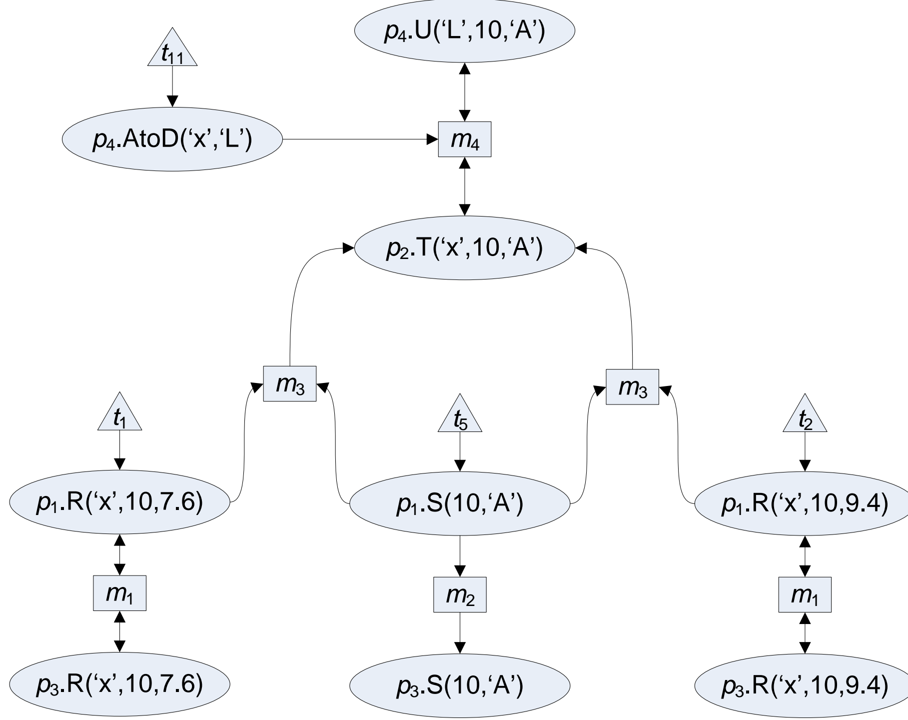


Figure 2.3: Provenance graph for a subset of our running example. Oval nodes are tuples, rectangular nodes are mappings, and triangles are token nodes indicating base data.

not present, and therefore the provenance expression $m_3(\mathbf{z_1} \cdot \mathbf{z_5})$ would also evaluate to not present. However, $\mathbf{z_{14}}$ would still be present, since it also contains the term $m_3(\mathbf{z_2} \cdot \mathbf{z_5})$, and p_2 does trust $\mathbf{z_1}$. Therefore $p_2.\mathtt{T}(\text{'x'}, 10, \text{'A'})$ would appear in p_2's instance.

It is relatively trivial to also use the provenance information for incremental maintenance. Suppose we have the example instance shown above. If p_1 then deletes $p_1.\mathtt{R}(\text{'x'}, 10, 7.6)$, this has the effect of setting $\mathbf{z_1}$ to being not present, which would also set the $m_3(\mathbf{z_1} \cdot \mathbf{z_3})$ component of $\mathbf{z_{14}}$ to being not present. However, there is an alternative derivation of $p_2.\mathtt{T}(\text{'x'}, 10, \text{'A'})$ that uses $\mathbf{z_2}$ instead, so that component of the provenance still evaluates to present and the tuple remains.

A more interesting case is if p_1 deletes both $p_1.\mathtt{R}(\text{'x'}, 10, 7.6)$ and $p_1.\mathtt{R}(\text{'x'}, 10, 9.4)$. Then, both $\mathbf{z_1}$ and $\mathbf{z_2}$ become not present. However, $\mathbf{z_{14}}$ still has a non-empty provenance, so it is necessary to consider whether $\mathbf{z_{24}}$ and $\mathbf{z_{11}}$ are still present. However, evaluating $\mathbf{z_{24}}$ means that we must evaluate $\mathbf{z_{14}}$, which is still unknown. Here, the situation is complicated by cyclic mappings (actually bidirectional mappings, but this is equivalent to a pair of unidirectional mappings that form a cycle). The question of whether a tuple is still present boils down to whether there is still a path from base tu-

ples to the derived tuple that satisfies certain constraints. This graph is a special case of the AND-OR graph commonly studied in computer science. The mapping nodes are AND nodes, where all of the inputs must be still derivable for the node to be derivable. The tuple nodes are OR nodes, meaning that at least one input must be derivable for the tuple to be derivable. All derivations must start from the provenance tokens attached to base tuples. Maintenance in the presence of cyclic mappings can be though of as garbage collection, where tuples with no paths to garbage collection roots (the base tuples) are deleted. The actual methods used for this are not germane to this thesis, but are detailed in Green et al. (2007a). Should a later update make provide a new path from base tuples to a derived tuple, then the derived tuple would be reinserted, just as if it were a truly new tuple.

In the interests of clarity, in the presentation here there were no *existentially* quantified variables in the mappings. ORCHESTRA does support such mappings, which arise naturally in many situations, such as in the example if mapping m_3 were inverted. If data were propagated from p_2 to p_1, the e value in $p_1.R$ would need to come from somewhere. Additionally, the introduction of existentials can cause infinitely-sized data instances if system semantics are not carefully defined. A discussion of these and related issues is beyond the scope of this thesis; an interested reader is referred to the discussion in Green et al. (2007a).

Internal System Structure

As described above, updates propagate through a network of mappings, subject to satisfaction of trust conditions. However, in this slightly naïve presentation, there is no clean way for a participant to delete a tuple that arrives at it through the mappings. A deletion would simply cause the mapping to be temporarily unsatisfied, and the tuple would reappear at the next reconciliation.

As described in Green et al. (2007a), each logical relation R in an ORCHESTRA instance is represented internally by four relations. The relations are shown in Figure 2.4. Briefly, they are:

- R^i, the *input table*. All mappings that logically place data in R are rewritten to instead place data in here.
- R^ℓ, the *local contributions table* of the peer that owns R. This contains the tuples inserted by that peer, which may later be removed, but only by the same peer.

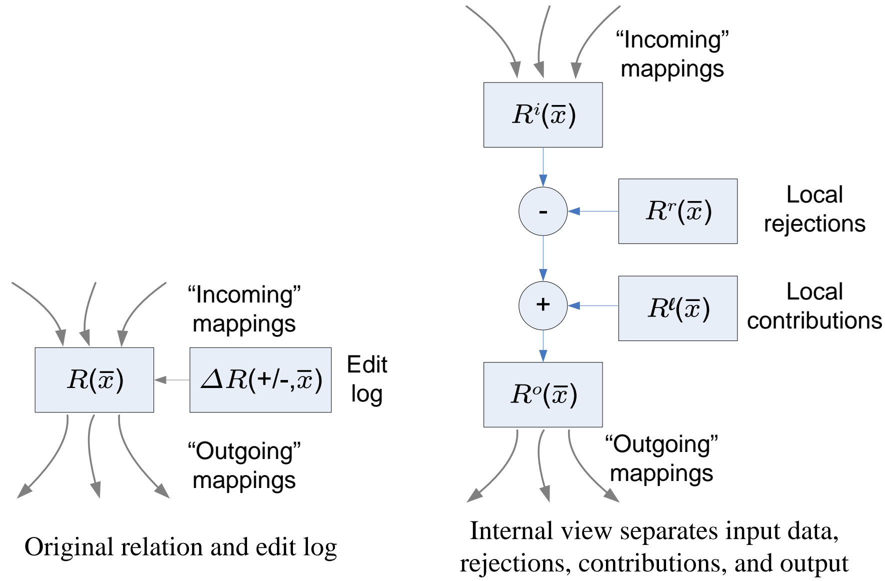


Figure 2.4: Internal CDSS representation of a relation turns each relation into four separate relations. Figure from Green et al. (2007a).

- R^r, the *rejections table* of the peer that owns R. These are tuples that arrive at R^i through mappings, but which the peer has decided are incorrect.
- R^o, the *output table*. This is the result of combining the effects of R^i, R^ℓ, and R^r using the mappings i_R and ℓ_R below. This table is used as the input to all mappings that read data from R, and is also the local instance used to answer queries posed by the peer that owns R.

The mappings that relate these internal tables are:

$$R^i(\bar{x}) \land \neg R^r(\bar{x}) \rightarrow R^o(\bar{x}) \qquad (i_R)$$

$$R^\ell(\bar{x}) \rightarrow R^o(\bar{x}) \qquad (\ell_R)$$

The use of different tables is an elegant way to satisfy both traditional data exchange consistency semantics and at the same time provide the necessary feature of being able delete undesired data.

2.3 Cloud Substrate for Orchestra

This chapter has motivated the collaborative data sharing model, and described at a high level the Orchestra CDSS that many people at the University of Pennsylvania have worked hard to design and

implement. We described how the system generates queries to translate updates between schemas, encode provenance, and determine which data items are trusted. All of this work assumes that there is a scalable storage system to hold the participants' relations and updates, and a scalable query processor to execute the queries the system generates.

The main focus of this thesis is this query processor and storage system. We describe how to build the substrate necessary to enable high-performance, high-availability storage and data transformation. The system is distributed, to make it scalable, and peer-to-peer, to make it easy to set up and maintain. We also consider the presence of transactions and conflicting updates, which were not covered in the work summarized in this chapter.

Recall that Figure 2.2 shows the major components of ORCHESTRA. This thesis details the design and implementation of these interacting components.

Publication As discussed in this chapter, publication shares a participant p's updates with the rest of the system. This involves updating p's update log in the *reliable versioned storage*, described below.

Update exchange We presented an overview of update exchange in this chapter. A more complete discussion is found in Green et al. (2007a). Update exchange poses queries over the published data in the *reliable versioned storage*, using our *distributed query execution* engine. These queries translate updates into the reconciling participant p's schema. They are then input to the *reconciliation* process, outlined below.

Reconciliation Reconciliation, the process by which p chooses a consistent subset of newly available transactions to apply, was briefly mentioned in this chapter. Reconciliation receives translated updates from the *update exchange* layer, supplemented with information about transactions and conflicting updates from queries posed over the *distributed query execution* layer. Based on user preferences, a subset of the translated transactions are applied to p's instance, which respecting transactional atomicity and dependencies. Reconciliation is the focus of Chapter 3 of this thesis, which describes the semantics of our reconciliation procedure and gives efficient algorithms for its implementation.

Distributed query execution Both *reconciliation* and *update exchange* depend on distributed query processing to speed their execution. Chapter 3 describes a prototype of a specialized dis-

tributed query execution system for reconciliation. Chapter 4 builds on these ideas to develop a general-purpose, distributed query execution engine, with an emphasis on reliability, that can be used to execute the queries generated by both reconciliation and update exchange. This query processor executes over the *reliable distributed versioned storage*, which we overview below.

Reliable distributed versioned storage The *distributed query execution* engine described above requires reliable access to mutable data in order to support the query processing needs of *update exchange* and *reconciliation*. The *data storage* layer described below only provides best-effort consistency. As described in Chapter 4, we use an indexing scheme to ensure that queries, posed relative to a particular version of the distributed database, execute over precisely that version of the database.

Data storage, partitioning, and distributed lookup We built the *reliable distributed versioned storage* layer over a simpler data storage and partitioning layer. We started with a well-known peer-to-peer data structure, the Distributed Hash Table (DHT), because of its self-configuring nature, native support for replication of data, and well-studied properties. However, DHTs are traditionally used in much larger settings, with tens or hundreds of thousands, or even millions, of nodes. Many design decisions made in traditional DHTs reflect this. In Chapters 4 and 5, we explore ways to tweak the DHT model in order to improve query performance and load balancing over the smaller "clouds" of nodes that Orchestra uses.

Together with the work reviewed in this chapter, the techniques presented in this thesis enable a complete Orchestra implementation. We increase the power of the Orchestra data model by adding integrity constraints. We show that it is possible to create a high-performance, reliable Orchestra implementation, and that we can do so using a cloud of nodes instead of centralized hardware. In short, this thesis shows that Orchestra is *feasible* in addition to being *interesting* and *possible*.

Chapter 3

Reconciliation: Algorithms and Implementation

Chapters 1 and 2 of this thesis introduced ORCHESTRA, a collaborative data sharing system which has been under development at the University of Pennsylvania for more than five years. They also described in some detail the overall operation of the ORCHESTRA system. However, they did not discuss issues of transactional atomicity, dependencies between updates, integrity constraints, and conflicting data. In this chapter, we detail the *reconciliation* process, which chooses a maximal set of compatible transactions to apply when a participant synchronizes with an ORCHESTRA instance. We also discuss ways of storing and retrieving those transactions.

This chapter is structured as follows:

- Section 3.1 reiterates the portions of the collaborative data sharing model that are relevant to reconciliation, and gives the rationale behind our approach to reconciliation.
- Section 3.2 defines the semantics of reconciliation, first for a simpler case without update or deletion, and then for the more complex case with modification. Our formalization emphasizes local autonomy, making forward progress, and providing intuitive behavior. We support *coexistence* of multiple instances, with consistency defined by *trust policies*, *compatibility*, and *transactional dependency*.
- Section 3.3 provides algorithms to implement the semantics defined in Section 3.2. It also introduces the *update store* abstraction, which isolates the reconciliation algorithm from the

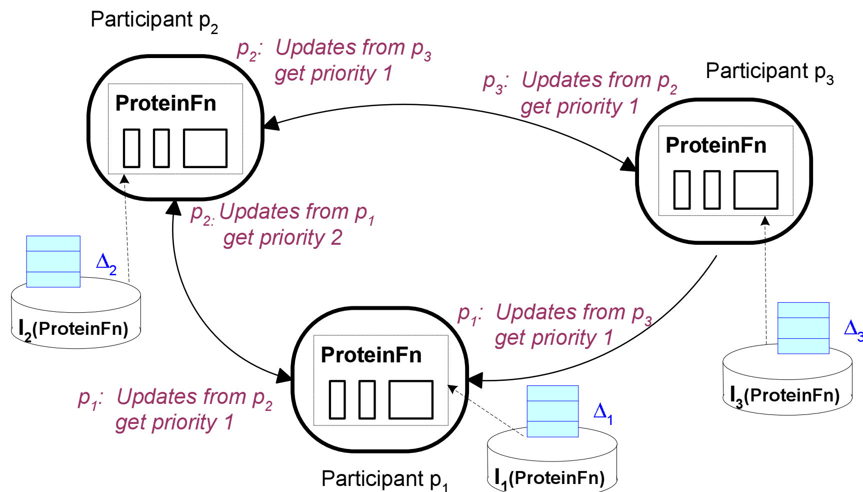


Figure 3.1: Collaborative data sharing system setting with three bioinformatics data warehouse participants sharing data on protein functions

storage and retrieval of updates.

- Section 3.4 shows how to provide two implementations of the update store. One uses a server-based RDBMS, and the other uses a custom peer-to-peer storage and computation layer. The latter was the inspiration for the general-purpose peer-to-peer work described in Chapter 4.

- Section 3.5 gives a performance analysis of these algorithms and update store implementations on bioinformatics-based workloads.

This chapter defines a model and methodology for reconciliation when all participants share a *single* database schema, in the presence of transactions, disagreement, and trust relationships. The complementary problem of translating updates across schemas in order to incrementally maintain data instances was discussed in Chapter 2 of this thesis. In the full ORCHESTRA implementation, the reconciliation procedures are identical, as updates are translated into the reconciling participant's schema before being considered by the reconciliation process.

3.1 CDSS Properties

An ORCHESTRA system, such as the example shown in Figure 3.1, consists of a number of collaborating participants ($p_1, \ldots, p_3$ in the figure), each of whom controls and edits its own data instance (denoted $I_1, \ldots, I_3$), and each of whom has a policy about what external data it is willing to trust

and accept (labels on the arcs). As data is modified at different participants (denoted $\Delta_1, \ldots, \Delta_3$), Orchestra publishes and propagates the updates to all participants that are willing to accept them. Our end goal is to facilitate update propagation in situations with both disparate schemas (i.e., each participant may have a different schema, with some attributes that may not have corresponding aspects in the other schemas) as well as disparate instances (i.e., each participant may have tuples with values that may not exist in other participants' instances).

Recall from Chapters 1 and 2 that each participant has its own local data instance. The CDSS is responsible for translating updates from other participants, and choosing a subset of those to apply to participant's local instance, based on user preferences. As mentioned previously, this chapter assumes a single fixed schema for all participants; the techniques described here would be performed on the output of the update exchange procedure described in Chapter 2 in a full Orchestra implementation. In Figure 3.1, therefore, all participants (p_1, p_2, p_3) share the same database schema ProteinFn. They may of course trust different data, and so their instances will not necessarily be the same. They publish their changes $(\Delta_1, \Delta_2, \Delta_3)$ to the CDSS, which uses them to maintain their instances (I_1, I_2, I_3). Finally, a series of *acceptance rules* or *trust conditions* (shown as the labels along the arcs between participants) define, for each participant, a *trust priority level* for updates from other participants. This may include rejecting some updates (or some entire other participants) out of hand, even if there is no conflicting data.

A central problem in a CDSS is determining the subset of updates that a participant should apply. We term this problem *reconciliation*: given the acceptance rules and updates published by participants, the reconciliation operation determines which updates should be applied to ("accepted by") the *reconciling participant* p. All updates that satisfy the acceptance rules and do not mutually conflict (or conflict with existing state) should be accepted; for conflicting updates, priorities are used to determine which (if any) updates are to be applied. Figure 2.2 from Section 2.2 showed how reconciliation fits into Orchestra as a filter between the update exchange process and the application of translated updates to a participant's local instance. As mentioned previously, in this chapter we assume a single schema for all participants, so there is no need for the complex query processing capabilities shown on the right side of that figure. We instead consider two simpler implementations of data storage and retrieval to provide the features needed from the right side of the figure. We will consider more general distributed data storage and query processing in Chapters 4 and 5.

We adopt a fairly simple conflict model, which defines a conflicting update as any update that (1)

results in an instance that is inconsistent with its integrity constraints, when applied applied to the current database instance, or (2) is *mutually incompatible* with some other published update that also satisfies an acceptance rule. We define later precisely what is meant by mutual incompatibility. It includes simple problems, like two updates that together cause a constraint violation (such as inserting two different values for the same key), and more complex problems, like two different modifications to the same tuple. The CDSS takes the idea of the peer data management system (Halevy et al., 2003) to the logical next step: in a PDMS, each participant has full control of its virtual mediated schema, which has *schema mappings* to other participants. Here, the participant also has control of its own *data instance*, including whether certain external updates should be applied, and what the preference between them should be, based on the trust conditions.

We assume that reconciliation is an operation that is done periodically, but not in real-time, by each participant: the participant will accept and apply a subset of all "recently published" updates to its data instance. Note that reconciliation is a matter of *importing* data, and therefore it can be done more or less frequently than publishing, though we assume that the two are performed together.

Informally, the goal of the CDSS is to, on demand from participant p_i, recompute instance $I_i(\Sigma)$, such that its contents satisfy the constraints imposed by the acceptance rules and the schema mappings, given the data instances in $I(\Sigma)$ and the Δ update operations published since $I_i(\Sigma)$ was last reconciled. For the setting we address in this chapter, all schema and update translation mappings are identity mappings, since every participant shares the same schema. In the next section, we more formally define what it means to for an instance of a CDSS to satisfy the acceptance rules, and we develop a semantics for reconciliation.

Consistency Model

Orchestra departs in a major way from other consistency models. While we put off a complete discussion of related work in this area until Chapter 6, we observe that most other work on consistency and concurrency attempts to create a transaction order that is *serializable*. Transactions that violate serializability are either rolled back and restarted (if possible) or simply undone, in order to restore consistency. Given that database instances are allowed (even expected) to diverge, traditional serializability makes less sense in this context. The goal of serializability is to ensure that the global instance is one that could have been reached if all transactions were applied, sequentially, to the same

instance. If there is no global instance, this goal is somewhat ill-defined.

As described previously, participants periodically publish and reconcile to export and import changes, respectively. We assume a global ordering on these operations, and term each such step an *epoch*. At each reconciliation, a single participant imports updates from outside. While it may also publish its own updates, no other participant will receive these until *it* reconciles. Thus, information flow is inherently (1) relative, as the updates participant p sees from participant q are those published since p last reconciled, and (2) asymmetric, as q will not immediately receive p's updates. Moreover, a reconciling participant has no way of knowing whether the updates it "sees" now will be revised in the future, or whether some other participant will publish a conflicting update in the future.

Since asymmetric dataflow is a natural consequence of the way we require the system to operate, we adopt a *dataflow-centric* consistency model, rather than one based on global consistency. We consider data items to flow from participant to participant as they are created, modified, and deleted. We then ensure that, if a transaction is applied, the transactions upon which it depends are also present. This ensures that related portions of the global instance are kept synchronized *for certain subsets of participants*, but that the subsets are allowed to diverge; any participant can leave one of these subsets and start a new, related one at any time. This flexibility is critical for our goal of allowing local autonomy while allowing data to flow between participants. We initially do not track reads, and instead only ensure that dependencies are satisfied for modifications and deletions; read tracking is necessary only if derived data is inserted into the database. Read tracking fits nicely into our framework, however, and we show later how it can be added.

Other Preliminaries

Before providing an example of reconciliation in a CDSS, we begin with a description of our high-level goals and basic approach. We first provide an overview of the goals, giving the rationale behind them, and then explain how these guide our strategy.

Monotonicity Once an update has been accepted by a participant, a future reconciliation may result in *changing* the results of the update, but the update itself will not be rolled back from the data instance. We feel that rolling back accepted updates (whose effects have been seen by the user) could be very confusing.

Trust policies In many bioinformatics and other settings, some sources are known to be more credible than others. We allow for each participant to provide a partial ranking of authority for such cases, allowing the system to automatically resolve certain conflicts. This can greatly reduce the need for user intervention.

Maximal progress Each reconciliation should make maximal use of all published updates available at the time. Since reconciliation only happens occasionally, it might otherwise be a very long time before the effects of published updates become available.

Least interaction Update sequences made at participant p should not interact with update sequences made at participant q in unexpected ways: in particular, if q makes a modification that conflicts with p, but revises its modification so it no longer conflicts, *before* p imports its changes, then p should consider q's update sequence to be compatible. We feel that transient conflicts, such as this, that are never observed by a participant, do not generally have semantic meaning, and can therefore be safely ignored. This is very similar to the *log cleaning* operation of IceCube (Kermarrec et al., 2001), which removes intermediate state to avoid temporary conflicts.

We now describe how each of these principles guides the functionality of the CDSS, before we present an example.

Monotonicity. We believe that a causality model in which every participant makes maximal progress based on what it has seen, and never "changes its mind," has several desirable properties. If a participant applies an update, then the results of that update remain in its instance until another participant explicitly changes them; updates never simply "vanish" because a competing transaction has a higher priority. If transactions contain multiple updates, and there are many conflicting transactions, then a single decision about which of the conflicting updates to accept can have many ramifications. If past decisions could be reconsidered, then a large portion of the database, seemingly unrelated, might be affected as a consequence the change. We were eager to avoid this potentially confusing behavior.

Trust policies. *Acceptance rules* of the form $\langle \textit{predicate}, \textit{priority} \in \mathbb{N} \rangle$ assign a positive integer priority level to a set of updates, based on predicates over the *content* as well as the *origin* of these updates. Larger numbers denote higher priority, and by default all updates are not trusted (priority level zero).

When multiple updates have equivalent (and highest) priority, our semantics is to adopt a "certain answers," open-world model in which *none* of the conflicting updates will be applied until a user intervenes.

Consider the relation *F(organism, protein, function)* that will be a running example throughout this chapter (the compound key consists of the attributes *organism* and *protein*). Participant p_1 might not trust p_2 at all, p_3 in general at priority 10 but at priority 20 when *organism* = 'mouse', and p_4 only when *protein* = *KIF4* at priority 10. When there are conflicting updates, a participant will prefer a higher priority one over a lower priority update. For example, p_1 would prefer the insertion of *F(mouse, KIF4, spindle stabilization)* by p_3 to the insertion of *F(mouse, KIF4, chromosomal positioning)* by p_4, since the former is given priority 20 and the latter priority 10. Continuing this example, the system cannot automatically choose between the insertion of *F(chicken, KIF4, spindle stabilization)* by p_3 to the insertion of *F(chicken, KIF4, chromosomal positioning)* by p_4, since they are both given priority 10, and neither is applied, pending manual intervention.

In general, we assume that the trust policies are correct and immutable from the moment a participant joins the system. In practice, it is clearly preferable that they be mutable. A variety of semantics are possible in this situation, including applying the new trust conditions retroactively. A full discussion of them is beyond the scope of this thesis, but is an interesting avenue for future exploration.

Maximal progress. When the partial ordering given by the trust policies is not sufficient to resolve a conflict, we continue to make progress using what we term *deferral*. If several updates conflict and the participant has not indicated a preference between them, the system will mark them as being deferred until a user resolves the conflict. Any future updates that might conflict with an unresolved conflict are themselves deferred — ensuring that the user does not inadvertently render them inapplicable. At some later time, the user can decide between them, and the system will reach the same state as if the decision had been made immediately. The participant can continue to update, publish, query, and reconcile in the interim, and all non-effected portions of the database will continue to be synchronized.

Least interaction. The mode of information exchange (and hence the cause of dependencies occurring among updates) will solely be via acceptance of updates from other participants. Since two participants may reconcile at different frequencies, we believe that any *intermediate* states of tuples

should not interact, i.e., if different participants make successive modifications to a tuple while not in contact with one another, any intermediate states should be disregarded, and only the final updates should be considered. As mentioned above, we consider strict serializability to be overly restrictive. Instead of attempting to serialize all transactions between participants when reconciling, we consider the effects of a subset of transactions *visible to the reconciling participant*.

In order to support acceptance predicates over update origins, we assume that every update is annotated with its provenance. In general, as described in Chapter 2, a single update may be the result of operations at multiple participants, and therefore our implementation uses the semiring provenance of Section 2.2. If all participants have the same schema, then each update comes from precisely one participant, and identifying that participant is sufficient; we use this for the remaining of this chapter. This model resembles the Information Source Tracking method of Sadri (1994) and the multi-viewpoint formalism of Ives et al. (2005).

CDSS Example

We now give an example of CDSS operation, to show how ORCHESTRA works at a high level, before formalizing the semantics of reconciliation. We refer to the example CDSS in Figure 3.1, where participant p_1 has a policy to accept update sequences from either p_2 or p_3, assigning them equal priority. In contrast, p_2 prefers updates from p_1 versus p_3, and p_3 only accepts updates from p_2. An exception to this rule, which we describe later, is that p_2 may make revisions to updates that originated from p_1 — in this case, p_3 must *transitively* accept this portion of p_1's data.

Notation. In this chapter we describe all updates in terms of *changes to values*, and annotate them with the identifier of a single originating participant. We consider the following operations: insert tuple (denoted $+R(\bar{a}; i)$ for an insertion of the tuple $\bar{a}$ by participant i into some relation R with a schema compliant with $\bar{a}$); delete tuple ($-R(\bar{a}; i)$); modify tuple ($R(\bar{a} \to \bar{a}'; i)$, where $\bar{a}'$ is a new set of attribute values conforming to schema of R). We also assume that updates may be grouped into transactions, denoted $X_{i:j}$, where i represents the identity of the originator of the transaction, and j represents its unique local transaction identifier. We assume that transaction identifiers are assigned in increasing order.

Figure 3.2 illustrates reconciliation over four epochs within this CDSS, for a single relation *F(organism, protein, function)*, where *(organism, protein)* is a key. At time 0, each participant p_i's in-

Epoch	**Participant** p_3	**Participant** p_2	**Participant** p_1
0	$I_3(F)\|_0$: {}	$I_2(F)\|_0$: {}	$I_1(F)\|_0$: {}
1	$X_{3:0}$ {+F(*rat,prot1,cell-metab*; 3)} $X_{3:1}$: {F(*rat,prot1,cell-metab* → *rat,prot1,immune*;3)} **publish and reconcile** $I_3(F)\|_1$: {(*rat,prot1,immune*)}		
2		$X_{2:0}$: {+F(*mouse,prot2,immune*; 2)} $X_{2:1}$: {+F(*rat,prot1,cell-resp*; 2)} **publish and reconcile** $I_2(F)\|_2$: {(*mouse,prot2,immune*), (*rat,prot1,cell-resp*)}	
3	**reconcile** $I_3(F)\|_3$: {(*mouse,prot2,immune*), (*rat,prot1,immune*)}		
4			**reconcile** $I_1(F)\|_4$: {F(*mouse,prot2,immune*)} {$X_{3:0}$, $X_{3:1}$, $X_{2:1}$} **deferred**

Figure 3.2: Reconciliation of F(*organism,protein,function*), with key (*organism,protein*), among the participants of Figure 3.1, over four epochs. Each participant chooses the transactions to apply when it publishes and reconciles according to the policies in Figure 3.1. The resulting instance for each epoch e is denoted with $I_i(F)|_e$. When transactions conflict, the participant always picks its own version first, or else the highest-priority one and its antecedents (if this is unique). It *defers* any transactions that have no unique "winner."

stance of this relation, denoted $I_i(F)|_0$, is empty. In epoch 1, participant p_3 applies two transactions (one of which revises the other), and then it publishes and reconciles its data. Since no other updates have been published, p_e ends Epoch 1 with state $I_3(F)|_1$, obtained by applying its own update sequence.

In the next epoch, participant p_2 introduces two new tuples and then reconciles. Its resulting state, $I_2(F)|_2$, is the result of applying its own updates. Although p_3 published two updates that p_2 trusts, these updates conflict with p_2's own updates — hence, it rejects them. In Epoch 3, p_3 reconciles a second time. Now it applies the *mouse* update from p_2; it rejects the *rat* tuple that is incompatible with its own local state. Finally, in the last epoch, p_1 reconciles. It trusts p_3 and p_2 equally. Hence, it accepts the non-conflicting *mouse* updates, but it must *defer* the remaining *rat* update transactions because they all conflict.

Given our intuitions from the basic principles and the preceding example, we now proceed to define a formal semantics for reconciliation.

3.2 Reconciliation

We begin by specifying the collaborative data sharing system formally. Recall that, for purposes of this chapter, we will define the CDSS for a setting in which all participants share a single schema.

Definition 1 (Collaborative data sharing system). *A* collaborative data sharing system *(CDSS) includes the following components:*

- Σ*, a schema representing the relations in the system.*
- P*, a set of participants,* $\{p_1, \ldots, p_n\}$.
- $\mathcal{A}$*, a mapping from each* $p_i \in P$ *to a set of* acceptance rules*, each of which is a pair* (θ, ν) *where* θ *is a predicate on updates in* Δ *over some relation* R *and* ν *is an integer priority that* p_i *assigns to tuples satisfying* θ.
- Δ*, a sequence of transactions of updates of the form* $+R(\bar{x}; i)$*,* $-R(\bar{x}; i)$*,* $R(\bar{x} \rightarrow \bar{x}'; i)$*, over each relation* R *and published by each participant* p_i.
- $I(\Sigma) = \{I_1(\Sigma), \ldots, I_i(\Sigma), \ldots, I_n(\Sigma)\}$*, the public database instances controlled by each* p_i.

- e*, an integer clock or* reconciliation epoch *counter. It is incremented each time a different participant publishes data. We assume that the first publication or reconciliation step defines the beginning of epoch 1. We denote the subset of* Δ *published in epoch* e *as* $\Delta|_e$.

Suppose we are given a CDSS as in Definition 1. Let us denote an update made to relation R as δ_R. We define the *priority relative to participant* p_i *of a transaction* X, $pri_i(X)$, as follows:

- 0, if any $\delta \in X$ is untrusted, i.e., there is no $(\theta, v) \in A(p_i)$ such that $\theta(\delta)$ is satisfied and $v > 0$.

- $max(\{v \mid (\theta, v) \in A(p_i) \wedge \theta(\delta) \wedge \delta \in X\})$, otherwise.

This is done to ensure that accepted transactions are fully trusted, and to favor the highest-priority updates over all others. Other methods of determining transaction priority from update priorities are of course possible, and may be more appropriate for certain applications. An obvious alternative is to use the minimum priority update in a transaction. The techniques presented here depend on a consistent method of determining transaction priority, but are not sensitive to precisely how it is done.

We say that two updates δ_R, δ'_R conflict (denoted $\delta_R \not\rightleftharpoons \delta'_R$) iff

- δ_R, δ'_R are both insertion operations with the same values for their key attributes, but different values for at least one other attribute, or

- one of δ_R, δ'_R is a deletion and the other is a replacement or insertion operation, and they have the same values for their key attributes, or

- δ_R, δ'_R are both replacement operations with the same source tuple value, where the replacement tuples have different values.

An update δ_R may also be incompatible with an instance I if applying δ_R to I would violate an integrity constraint, or modify or delete a value that is not present. We generalize this to say that two transactions X, X' conflict (denoted $X \not\rightleftharpoons X'$) iff an update $\delta \in X$ conflicts with an update $\delta' \in X'$, and that a transaction X is incompatible with an instance I iff an update $\delta \in X$ is incompatible with I.

Our model of conflicts is quite conservative, in that *any* two updates that result in different result tuples for the same input tuple are deemed conflicting. Without detailed knowledge of which attributes are independent, this is the best a system can do. For example, consider a relation *R(a,b,c)*,

where *a* is the key attribute, and *b* and *c* are truly independent. In this case the updates that replace the tuple $R(a, b, c)$ with $R(a, b', c)$ and $R(a, b, c')$ are considered by ORCHESTRA as conflicting. However, their effects could be combined as updating $R(a, b, c)$ to $R(a, b', c')$. The problem can in this case be solved by fully decomposing the schema into two relations, *RB(a,b)* and *RC(a,c)*. However, if the connection between *b* and *c* were deeper, this might not be the case. Studying the effects of schema design on system operation is an interesting area for future research, but will not be discussed further in this thesis.

We now consider two versions of the reconciliation problem. We begin with an insert-only setting, which is simpler and easier to understand but much less powerful. We then consider the general reconciliation problem, where tuples may be updated or deleted, as would happen in most real-world settings.

Insert-Only Reconciliation

In the insert-only case, every transaction in a given epoch can be considered independently; without modification of existing data, there are no (write) dependencies between transactions. An insertion may be applied so long as it does not conflict with a previously applied insertion, nor does it conflict with a transaction of equal or higher priority.

For any epoch e, let $\Delta_{acc}(i)|_e$ be the set of transactions from $\Delta|_e$ acceptable to p_i. We define $\Delta_{acc}(i)|_0$ to be the empty set, and for all other epochs let

$$\Delta_{acc}(i)|_e = \left\{ X \in \Delta|_e \text{ s.t. } \begin{array}{l} (\nexists X' \in \Delta|_e \text{ s.t. } X \not\Leftrightarrow X' \wedge pri_i(X') \geq pri_i(X)) \wedge \\ (\nexists X'' \in \Delta|_{e'} \text{ s.t. } e' < e \wedge X \not\Leftrightarrow X'') \end{array} \right\}$$

From this, it is straightforward to define the reconciliation problem for a given participant in the insert-only model.

Definition 2 (Insert-only reconciliation).
Let $I_i(R)|_e$ *be the instance of relation* R *at participant* p_i *in epoch* e. *The* insert-only reconciliation problem for participant p_i *is to compute* $I_i(R)|_e$ *for every* $R \in \Sigma$, *given some initial* $I_i(R)|_{e_0}$ *and* Δ *from* e_0 *to* e. *For each relation* R, *let* $Res = I_i(R|_{e_0})$. *A tuple* t *must appear in instance* $I_i(R)|_e$ *iff it appears in the instance* Res *resulting from recursively applying, for each* τ *from* e_0 *to* $e - 1$ *(in increasing order),* $Res = apply(\delta_R, Res)$ *for every* δ_R *in* $\Delta_{acc}(i)|_{(\tau+1)}$.

Insert-only reconciliation is very simple to compute algorithmically: during epoch τ, for each p_i, we simply consider each published transaction X in isolation and determine whether it is in $\Delta_{acc}(i)|_\tau$, meaning that it uniquely has the highest priority of any transaction with which it conflicts. If so, we apply the transaction.

Replacement and Deletion

If we allow for replacement and removal, the semantics of reconciliation must change significantly: now an update may *depend* on other, *antecedent* updates. For example, participant p_1 may insert the tuples *F(mouse, KIF5, spindle stabilization)* and *F(mouse, KIF4, chromosomal positioning)* in one transaction, and then update *F(mouse, KIF4, chromosomal positioning)* to *F(mouse, KIF6, chromosomal positioning)* in another transaction. We say that the second transaction is dependent on the first one, since it modifies data from it. Since the tuples in the first transaction were inserted together, there is some higher-level meaning to the fact that they are both there, and they are in some sense related. If a second participant p_2 reconciles and receives these transactions, it should either apply the first one or both of them, since these represent states which p_1 experienced at some point in time.

This is a simple example, since there are only two transactions involved, which therefore form a linear chain. In general, though, the dependencies between transactions form a directed, acyclic graph. We think of the edges in the graph as indicating dataflow, so an edge from transaction t_1 to transaction t_2 indicated that t_1 is an antecedent of t_2. A transaction may depend on multiple previous transactions, which in turn may depend on additional, possibly overlapping, sets of transactions. In order to apply a transaction, we require that all of its immediate and transitive antecedent transactions have been applied first. Therefore, for a given set of related transactions (i.e. transactions that have at least one transitive antecedent in common, or equivalently a connected component in the transaction dependency DAG), a participant must accept a (possibly empty) subset of them, such that each accepted transaction does not depend on any unaccepted ones. We think of the *terminal* accepted transactions, those upon which no other accepted transactions depend, as forming the *frontier* of acceptance. Since no transactions are accepted and then later rejected, this frontier is always expanding away from the *root* transactions (those with no dependencies).

Figure 3.3 shows an example transaction dependency graph for our running example schema. The set $\{t_1, t_2, t_3\}$ forms a valid set of accepted transactions, with t_2 and t_3 being the frontier. The

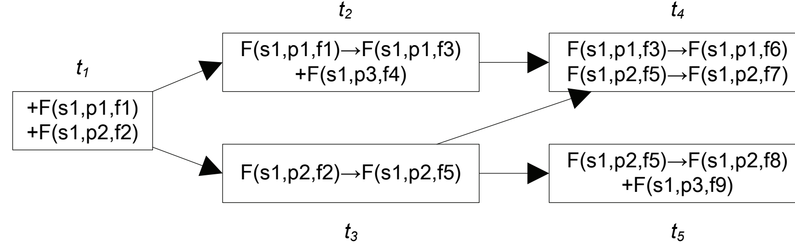


Figure 3.3: Example transaction dependency graph

set $\{t_1, t_2, t_4\}$ is not a valid set of transactions to accept, since t_4 depends on t_3. $\{t_1\}$, $\{t_1, t_2, t_3, t_4\}$ and $\{t_1, t_3, t_5\}$ are also valid such sets. The set of all the transactions in the graph and $\{t_1, t_2, t_3, t_5\}$ are also valid from a dependency point of view; however, since t_2 and t_5 conflict, they cannot both be accepted by a single participant (though later we will see that if a later transaction removes the conflict between the two transactions they then could coexist).

These antecedent updates may have originated from participants *other than* the participant who published the most recent update, and the original source of that update might not itself be trusted by the participant who is reconciling. In our above example this must have been the case. Perhaps p_1 performed t_1, t_2, t_3, and t_4 and p_2 performed t_5, or perhaps all transactions came form different participants. Regardless, different pieces of the transaction dependency graph may originate from different participants.

As mentioned previously, in a reconciliation model with deletions and updates, one transaction may introduce a conflict, but a succeeding transaction may remove that conflict. For instance, to continue the example of Figure 3.2, suppose that in epoch 2, participant p_3 first introduced a sequence of transactions:

$$X_{3:2} : \{+F(\textit{mouse,prot2,cell-resp})\}$$

$$X_{3:3} : \{F(\textit{mouse,prot2,cell-resp}) \rightarrow (\textit{mouse,prot3,cell-resp})\}$$

where initially the wrong protein was given the function *cell-resp*. In this case, while transaction $X_{3:2}$ clearly conflicts with $X_{2:0}$, intuitively p_3 should accept $X_{2:0}$, since this does not conflict with its state after applying the full transaction sequence above. In general, given a transaction sequence, one can take the constituent update sequence and "flatten" it into a set of direct updates by removing intermediate steps, as described in Ghandeharizadeh et al. (1996) and Ives et al. (2005). The sequence $[X_{3:2}, X_{3:3}]$ above can be minimized to $\{+F(\textit{mouse,prot3,cell-resp})\}$.

Let *applied*(p_i, e) be the set of updates that have been applied by participant p_i up through epoch e. Also, for a transaction X published in epoch e, we define the *antecedent* set, *ante*(X), to contain any transaction $X' \in \Delta_\tau$, $1 \leq \tau \leq e$, where X' either inserts a new tuple, or makes a modification to a tuple, which X directly deletes or modifies.

Definition 3 (Transaction extension). *We define* p_i*'s* transaction extension *of transaction* X*, reconciled in epoch* e*, to be the transitive closure of* X*'s antecedents, so long as those transactions have not yet been accepted by* p_i *in epoch* e*. We denote this transaction extension as* $\mathtt{te}_{i|e}(X)$*, and sort the transactions by their order in* Δ*.* $\mathtt{te}_{i|e}(X)$ *therefore contains the sequence of transaction identifiers that* p_i *needs to apply at epoch* e *in order to apply transaction* X *while respecting the dependency rules described above. We say that* X *is the root of its transaction extension.*

Definition 4 (Update footprint). *Given a list of transactions* L*, sorted by the order of their application, we can define their* update footprint *to be the concatenation of the updates in those transactions as*

$$\mathit{uf}(L) = [\delta \in X \textit{ for each } X \in L]$$

Definition 5 (Update extension). *Now, given a function flatten*(s) *which flattens a sequence of updates into a set of mutually independent updates, as described above, we can compute the* update extension *of transaction* X *for participant* p_i *at epoch* e *as*

$$U_{i,e}(X) = \mathit{flatten}\left(\mathit{uf}\left(\mathtt{te}_{i|e}\left(X\right)\right)\right)$$

As before, we say that X *is the root of this update extension.*

The update extension $U_{i,e}$ represents the set of changes need to apply transaction X to participant p_i's instance at epoch e, with all intermediate steps removed; recall that this eliminates transitory conflicts between transactions that are later removed. Since the update extension is the smallest unit of changes that can be applied to an participant's instance without violating the transaction dependency constraints, the reconciliation procedure updates at the granularity of update extensions. At a very high level, the reconciliation procedure needs to

1. retrieve all newly available, trusted transactions,
2. compute their update extensions, and

3. choose a set of non-conflicting update extensions to apply, while favoring high-priority update extensions.

Before defining precisely what this means, there are several cases where conflicts between update extensions are in fact not a problem. These occur when the transaction extensions used to create the update extensions intersect; in other words, the roots of the update extensions have unapplied, transitive antecedents in common.

Definition 6 (Subsumption). *We say* X subsumes *some other* X' *if its transaction extension is a superset of* X'*'s transaction extension.*

If the root X of one update extension $U_{i,e}(X)$ being considered subsumes another one $U_{i,e}(X')$, then clearly the system should apply either $U_{i,e}(X)$, $U_{i,e}(X')$, or neither of them. $U_{i,e}(X)$ should be favored, since it contains the effects of X', but if the update extension of X cannot be applied, then $U_{i,e}(X')$ should be considered.

Since the dependencies form a DAG and not a linear graph, however, potential interactions between update extensions are more complicated than subsumption and being completely disjoint. We therefore define the notion of a direct conflict between two update extensions.

Definition 7 (Direct conflict). *Two transactions* X, X' directly conflict *iff flatten* $\left(uf\left(te_{i|e}(X) - te_{i|e}(X')\right)\right)$ *conflicts with flatten* $\left(uf\left(te_{i|e}(X') - te_{i|e}(X)\right)\right)$*, or in other words, there is a conflict between their update extensions when* excluding *the transactions the two have in common.*

We only consider direct conflict to be a cause for an update extension to not be applied during reconciliation. Indirect conflicts are the result of the transaction dependency requirements, and while they may require some additional work during reconciliation to avoid repeated application of antecedent transactions (to avoid inserting the original and modified version of a tuple, say), they do not indicate any fundamental incompatibility.

There is one final point to be considered before defining the semantics of reconciliation. Up to now, we have defined priorities for individual updates and transactions, but we have not discussed priorities for update extensions. In general, a trusted transaction may depend on other transactions of higher priority, lower priority, or ones that are entirely untrusted. We assign the priority of its root to each update extension, which has the effect of favoring higher-priority transactions over all lower-priority transactions, even if there are many lower-priority transactions or those high-priority

transactions have untrusted antecedents. While this is a choice, it leads to a simple explanation for why transactions are accepted or not; we felt that transparency of operation was an important quality in a system such as ORCHESTRA.

We can now define the general reconciliation problem for updates that include deletions and replacements. A solution to the general reconciliation problem must also maintain information about whether prior reconciliation operations marked certain transactions as rejected or deferred. We must defer any transactions that depend on a deferred transaction, since we cannot yet tell if the transaction conflict with a participant's current instance. As presented here, reconciliation also rejects any transactions that depend on a rejected transaction, since otherwise the system would be "changing its mind" about whether to accept a transaction or not; additionally, transactions may have been explicitly rejected by a user when resolving a conflict between two transactions at the same priority. This approach leads to perhaps the simplest semantics, though others are certainly possible.

Definition 8 (General reconciliation). *We define the* general reconciliation *problem for participant* p_i *as follows. During epoch* e*, given an initial* $I_i|_{e_0}$*, sets of previously deferred transactions deferred*(p_i, e_0)*, previously accepted transactions accepted*(p_i, e_0)*, and previously rejected transactions rejected*(p_i, e_0)*, all* Δ_i *from* e_0 *to* e*, and the newly available, trusted transactions* $T \subseteq \{\Delta_{e_0} \cup \ldots \cup \Delta_e\}$ *(where all* $t \in T$ *are trusted by* p_i*), create sets accepted*(p_i, e)*, rejected*(p_i, e)*, and deferred*(p_i, e)*, subject to the following conditions:*

1. *accepted*$(p_i, e) \supseteq$ *accepted*(p_i, e_0)*, rejected*$(p_i, e) \supseteq$ *rejected*(p_i, e_0)*, and deferred*$(p_i, e) \supseteq$ *deferred*(p_i, e_0)
2. *The transaction extension* $te_{i|e}(X)$ *of a* $X \in \{T - \textit{rejected}(p_i, e)\}$ *must be contained in deferred*(p_i, e) *if*
    a) $U_{i,e}(X)$ *directly conflicts with some* $X' \in$ *deferred*(p_i, e)*,*
    b) $te_{i|e}(X) \cap \textit{deferred}(p_i, e) \neq \emptyset$*, or*
    c) $U_{i,e}(X)$ *conflicts with* $U_{i,e}(X')$ *for some* $X \in \{T - \textit{rejected}(p_i, e)\}$ *of equal priority.*
3. *A transaction* $X \in T$ *must be in rejected*(p_i, e) *if*
    a) $te_{i|e}(X) \cap \mathtt{rejected}(p_i, e) \neq \emptyset$*,*
    b) $U_{i,e}(X)$ *conflicts with* $U_{i,e}(X')$ *for some* $X' \in$ *accepted*(p_i, e) *of higher priority, or*

c) *applying* $U_{i,e}(X)$ *to* $I_i|_{e_0}$ *would either cause a constraint violation or attempt to modify or delete a value that is not present.*

4. *No transactions may be in more than one of rejected*(p_i, e)*, deferred*(p_i, e)*, accepted*(p_i, e). $T \subseteq \{$*rejected*$(p_i, e) \cup$ *deferred*$(p_i, e) \cup$ *accepted*$(p_i, e)\}$.

5. *Any* $X \in T$ *that is not is neither required to be in rejected*(p_i, e) *or deferred*(p_i, e) *due to the above rules (or conflict resolution, described in Section 3.2) must be in accepted*(p_i, e).

6. *For any transaction* $X \in$ *accepted*(p_i, e)*, all transitive antecedents of* X *must also be in accepted*(p_i, e).

and for every $X \in \{$*accepted*$(p_i, e) -$ *accepted*$(p_i, e_0)\}$ *(that is to say, every newly applied transaction), apply those transactions to* $I_i|_{e_0}$ *to produce* $I_i|_e$.

We now describe and motivate the items in the above definition. The first part of Definition 8 enforced monotonicity properties. Condition 1 simply says that no transaction that was accepted, rejected, or deferred during the previous reconciliation is affected by the reconciliation procedure.

The second part of Definition 8 describes situations under which transactions must be deferred. Conditions 2a and 2b defer any transactions whose potential to be accepted might be impacted by the decision a user makes about another deferred transaction; pending that user input the system cannot determine whether the transaction is compatible with the current database instance or not, or whether the transaction has an antecedent transaction that was rejected. Condition 2c ensures that all transactions that conflict with another transaction of the same priority are deferred; initially, when *deferred* is empty, this is only way for transactions to become deferred. It specifically excludes transactions that can be rejected out of hand, rendering the conflict moot. Note that these conditions defer all transactions in a trusted transaction's transaction extension, since all of these would need to be applied in order to apply that trusted transaction.

The third part of Definition 8 describes the three situations in which a transaction must be rejected. Condition 3a rejects all transactions that have a rejected antecedent. Condition 3b rejects all transactions that conflict with a transaction of higher priority that is being accepted, since the user has expressed a preference for the higher priority transaction and both cannot be applied simultaneously. Condition 3c rejects all transactions whose update extension cannot be applied to the participant's current instance without violating integrity constraints.

Condition 4 of Definition 8 states that transactions may be deferred, rejected, or accepted, but not more than one of the above, and that all trusted transactions for the current reconciliation must be in one of these sets. In other words, the reconciliation procedure must decide what to do with each trusted transaction.

Finally, condition 6 enforces that, if a transaction is accepted during the current reconciliation, its antecedents must also have been, either during the current reconciliation or during a previous one. More precisely, it enforces this condition for all transactions that have ever been accepted. However, inductively this must already have been the case for all transactions accepted during previous reconciliations. Since the set *accepted* grows monotonically, the constraint only affects transactions that might be accepted during the current reconciliation; it is already satisfied for earlier accepted transactions.

This definition hides a subtlety implicit in the definition of the transaction and update extensions of a transaction: they may change during the reconciliation process. If an antecedent transaction is applied in the process of applying another update extension (i.e. the transaction extension overlap), then a naive approach to reconciliation that simply applied a precomputed update extension might apply a transaction multiple times. Therefore, it is necessary for an implementation to ensure that each transaction is applied only once. We will revisit this in Section 3.3, where we discuss algorithms for reconciliation.

Read Dependencies

An extension to this model is to add read dependencies between transactions. We can do this with an additional operation, the so-called " check" operation, denoted by $?R(t)$. This operation says that the participant that created a transaction X read (but did not update) the tuple t. This adds the last transaction to modify t as an antecedent of the current transaction, enabling read dependency. This is still a somewhat relaxed definition of consistency. For a participant p to accept X, it is not required that t *still* be present in I_p, just that the transaction that inserted t have been accepted; another transaction may have modified it. We do not experimentally explore this extension. While there are cases where it may be useful, for such dependencies to show up in a database transaction log, the read and write must occur in the same transaction; this is unlikely to be the case for complex analyses. It would be interesting to explore this further for our target bioinformatics workloads.

An interesting case does, however, arise if we ensure read dependencies are satisfied, *and* after each transaction, the participant that performed it publishes it and all other participants then reconcile. If all participants trust each other fully, we then achieve complete serializability. As we have said before, we do not feel that in general serializability is either practical or desirable (due to the inherent inconsistency in collaborative data sharing). It is nevertheless reassuring to see that it appears at the limit of what a CDSS can do. We do not, however, see this mode of CDSS operation as being generally useful.

Conflict Resolution

Like many data management systems, ORCHESTRA makes as many decisions as possible automatically, and defers to the user when necessary. In ORCHESTRA's case, this means that it gives preference to higher priority transactions over lower priority ones, but needs user input to decide between conflicting transactions of the same priority. A key feature is that the system can continue to make progress even while it is not yet know whether some transactions will be accepted or not. Transactions whose acceptability depend on user decisions (directly or indirectly) will be deferred, along with the initial transactions of equal priority. All other transactions can be decided, since they are independent of the user decisions about past transactions.

Once a number of items have been deferred, the process of *conflict resolution* makes use of the solution to the reconciliation problem stated above. To resolve a conflict, the user specifies one or more trusted transactions to remove from the deferred set for some epoch e (and all subsequent epochs) and instead places them in the corresponding rejected sets; the deferred sets at and after e are cleared. Conflict resolution is specified in this negative way (the user specifies transactions to accept instead of transactions to reject) because it is always possible to respect a user's request that a transaction not be accepted; it may not be possible to respect a request that a particular transaction be accepted, due to constraint violations, deferred transactions, or rejected antecedents.

The reconciliation procedure is then rerun for e and all subsequent epochs. All decisions that were already made for a particular epoch will continue to stand, but it may be possible to decide that transactions that previously had to be deferred may be accepted or must be rejected, due to the smaller set of deferred transactions. When applying a transaction during an epoch $e' \geq e$ to $I_i|_{e'}$, it is also applied $I_i|_{e''}$ for all epochs $e'' > e'$. Clearly only transactions that are being newly decided

(and their transaction extensions) should be applied to instances.

Proposition 1 (Applicability). *The update extension* $U_{i,e}(X)$ *for any trusted transaction* $X \in$ *deferred*(p_i, e) *can be applied to any* $I_i|_{e'}$ *for any* $e' \geq e$ *without introducing constraint violations or attempting to modify non-present values.*

Proof. If X was not rejected in by the reconciliation algorithm in epoch e, then $U_{i,e}(X)$ could be applied to $I_i|_e$ without introducing constraint violations or attempting to modify non-present values, or the reconciliation algorithm would have been forced to reject it instead of deferring it. Since any modification in an epoch $e' \geq e$ that would have conflicted with $U_{i,e}(X)$ (and could therefore render it inapplicable) would also have been deferred, any $I_i|_{e'}$ must have the same state for all values created, modified, or deleted by $U_{i,e}(X)$. Therefore, since $U_{i,e}(X)$ could be applied to $I_i|_e$, it can also be applied to $I_i|_{e'}$. $\square$

From Proposition 1 and its proof, it should be clear how deferred transactions allow automatic updating of a participant's instance through reconciliation to continue while ensuring that deferred transactions can still be applied if and when a user makes a decision to do so. The more quickly a user resolves conflicts, the more database state is available, but the system will all reach the same state irregardless of the order in which conflicts are resolved.

We now move on to algorithms that perform the reconciliation procedure described in this section.

3.3 Reconciliation Algorithms

Definition 8 describes in a declarative fashion what a general reconciliation algorithm must do. When participant p_i decides to reconcile, it must retrieve the newly published, trusted transactions (the *relevant* transactions). It must compute their transaction and update extensions, and then chose a subset of those to apply, based on integrity constraint and user preferences. It then must update the participant's instance to reflect the transactions it has chosen to accept, and record the decisions it made. In this section, we describe concrete algorithms that satisfy this definition.

The computation described above can either be centralized or distributed. If the work is centralized on the reconciling participant, we call it *client-centric reconciliation*, since it is typically the reconciling participant that retrieves all of the relevant transactions and decides which to apply. An

	Client-Centric Reconciliation	Network-Centric Reconciliation
Distributed Store	Pros: No central store, medium communication Cons: Needs stable base of connected peers, reconciliation work all at one peer	Pros: No central store, distributed reconciliation work Cons: Highest communication, needs stable base of connected peers
Central Store	Pros: Low communication, high reliability Cons: Needs reliable central server, reconciliation work all at one peer	Pros: Distributes reconciliation work across many peers, high reliability Cons: High communication, needs reliable central server

Figure 3.4: Comparison of reconciliation algorithms and update stores

alternative is *network-centric reconciliation*, in which computation is distributed across the entire network of participants. While the network-centric approach would place less load on the reconciling participant by distributing almost all of the work across the network, the client-centric approach generates less network traffic, and it allows for a considerably simpler reconciliation algorithm. It also may allow potentially sensitive information, like the trust conditions, to be kept private from other participants.

In this thesis, we only consider client-centric reconciliation. Experimental evaluation, given in Section 3.5, showed that the cost of retrieving updates from storage (either local or distributed) was the dominating cost in reconciliation, so we did not consider it necessary to develop a distributed implementation of the reconciliation procedure.

The reconciliation algorithm needs to access several different kinds of data to perform the operations outlined above. It must access the log of published transactions, and the instance of the reconciling participant. It also needs to read and modify the sets of applied, rejected, and deferred transactions for the reconciling participants. We define an *update store* module to provide a general interface to much of the aforementioned state. We have explored using both a centralized server and a distributed store in which the participants themselves store the state.

Each combination of reconciliation algorithm and update store implementation has its own unique benefits, as shown in Figure 3.4. Our initial implementation uses client-centric reconciliation, which is considerably simpler both to understand and to implement; we couple that with either

central or distributed storage. As future work we propose to implement network-centric reconciliation over distributed storage using the distributed storage layer described in Chapter 4.

In order to implement an algorithm for the general reconciliation problem given in Definition 8, we introduce several new concepts:

Dirty values are key values that are modified (i.e. read or written) by a deferred transaction. As mentioned previously, any transaction that reads or writes a value whose key is in the dirty value set must be deferred, in order to ensure that a previously-deferred transaction can always be accepted later. They are used to enforce condition 2c from the definition to avoid performing many pairwise compatibility checks; instead a set can be maintained.

Conflict groups are groups of conflicts with the same type that involve the same key value; the reconciliation algorithm groups conflicts for each reconciliation into such groups.

Options are groups of transactions within a conflict group that make the same modification to the key value. At most one option can be accepted for each conflict group when conflicts are resolved; the transactions from the other groups are rejected.

In the common case, each option within a conflict group will have only one transaction in it. Consider our running example of the *F(organism, protein, function)* relation. Suppose we had three transactions

t_1 {+*F(mouse, KIF4, spindle stabilization)*, +*F(mouse, KIF5, spindle stabilization)*}

t_2 {+*F(mouse, KIF4, chromosomal positioning)*}

t_3 {+*F(mouse, KIF5, chromosomal positioning)*}

Here there would be two conflict groups, one for ⟨*mouse,KIF4*⟩ with options $\{\{t_1\}, \{t_2\}\}$, and one for ⟨*mouse,KIF5*⟩, with options $\{\{t_1\}, \{t_3\}\}$. These list the sets of compatible transactions for each key, and the user should choose one (or none) of those sets as valid; the others will be rejected in the conflict resolution procedure outlined in Section 3.2.

In a more complicated scenario, there may be multiple transactions within each option. If we slightly alter the above example to

t_1 {+*F(mouse, KIF4, spindle stabilization)*, +*F(mouse, KIF5, spindle stabilization)*}

t_2 {+*F(mouse, KIF4, chromosomal positioning)*}

t_3 {+*F(mouse, KIF4, chromosomal positioning)*}

then there would be one conflict group with two options, $\{\{t_1\}, \{t_2, t_3\}\}$.

We now present the client-side algorithm for reconciliation, the core of which is the RECONCILE-UPDATES procedure given in Algorithm 3.1. It determines which updates the participant can apply or reject during a particular reconciliation, and which it must defer, in a manner satisfying the requirements of Definition 8. It also assigns the deferred transactions into conflict groups, as described above, to explain to users which transactions were deferred and why. As described in Section 3.2, this algorithm is also run again after decisions for deferred transactions have been supplied by the user. Then, after recording the transactions a user has decided to reject, it reexamines the remaining deferred transactions to discover which, if any, can now be accepted.

Algorithm 3.1 RECONCILEUPDATES(*recno*)
Input: *recno* (reconciliation number to perform)
Helper functions are given as Algorithms 3.2, 3.3, 3.4, and 3.5.

```
1:  txns ← IDs of the undecided trusted transactions new for recno for this participant
2:  prio ← Mapping from index in txns to priority
3:  prios ← Set of all transaction priorities
4:  Sort prios in decreasing order
5:  for t ∈ txns do
6:     upEx[t] ← The flattened update extension of t
7:     decision[t] ← CHECKSTATE(recno,upEx[t])
8:  end for
9:  conflicts ← FINDCONFLICTS(txns, upEx)
10: for txnPrio ∈ prios do
11:    decision ← DOGROUP(txnPrio, conflicts,prio,decision)
12: end for
13: Record decision at recno
14: for t ∈ txns do
15:    if decision[t] = ACCEPT then
16:       Apply upEx[t]
17:    end if
18: end for
19: deferred ← {txn | decision[txn] = DEFER}
20: UPDATECONFLICTS(recno,deferred)
```

The RECONCILEUPDATES is given as Algorithm 3.1, and the various helper functions appear as Algorithms 3.2, 3.3, 3.4, and 3.5. RECONCILEUPDATES begins by computing the flattened update ex-

tension of each trusted transaction. The call to CHECKSTATE at line 7 determines which transactions much be rejected or deferred because of the reconciling participant's dirty value set or materialized state. The call to FINDCONFLICTS at line 9 discovers conflicts between the flattened update extensions of trusted transactions. The algorithm then calls DOGROUP at line 11 to consider each group of transactions with the same priority, in decreasing order of priority; the decreasing order allows the algorithm to proceed greedily and consider each group only once. Within each group, transactions that conflict with higher-priority accepted transactions are rejected, and those that conflict with higher-priority deferred transactions are themselves deferred; if conflicts are found between two non-rejected transactions within a group, both are deferred. Once all priority groups have been considered, RECONCILEUPDATES has made decisions for all trusted transactions. Line 13 records which transactions the participant has decided to accept or reject. Lines 14-18 update the state of the local database; it is necessary to recompute the update extension since the antecedents of the trusted transactions may overlap. Line 20 updates the participant's dirty value set and list of conflicts for the current reconciliation.

Algorithm 3.2 CHECKSTATE(*recno*,*upEx*)
Input: *recno* (reconciliation number), *upEx* (update extension of transaction)
Output: decision for input transaction

```
1: if upEx contains a value dirty at recno then
2:     return DEFER
3: else if upEx contains a rejected transaction then
4:     return REJECT
5: else if upEx is incompatible with the instance at recno then
6:     return REJECT
7: else
8:     return ACCEPT
9: end if
```

Suppose that during a particular reconciliation there are t relevant transactions, each of which has at most a undecided antecedents. Further suppose that each transaction contains at most u component updates. In this case, computing the flattened update extensions will take time $O(tua)$, since that much time is needed even to read through the updates for the relevant transactions. Checking for pairwise conflicts between the update extensions will take time at most $O\left(t^2 + tua\right)$, if a hash table-based conflict detection algorithm is used. This conflict detection step asymptotically dominates all other work done afterwards by the RECONCILEUPDATES procedure, giving a combined running time

Algorithm 3.3 FINDCONFLICTS(*txns*,*upEx*)
Input: *txns* (set of transaction IDs), *upEx* (map from transaction ID to flattened update extensions)
Output: Map from transaction ID to set of conflicting transaction IDs

1: *conflicts* $\leftarrow \emptyset$
2: **for** t, t′ $\in$ *txns* **do**
3:     **if** *upEx*[t] conflicts with *upEx*[t′] **then**
4:         **if** neither t nor t′ subsumes the other **then**
5:             *conflicts*[t] $\leftarrow$ *conflicts*[t] $\cup$ {t′}
6:             *conflicts*[t′] $\leftarrow$ *conflicts*[t′] $\cup$ {t}
7:         **end if**
8:     **end if**
9: **end for**
10: **return** *conflicts*

Algorithm 3.4 DOGROUP(*txnPrio*,*conflicts*,*prio*,*decision*)
Input: *txnPrio* (map from transaction ID to priority), *conflicts* (map from transaction ID to IDs of conflicting transactions), *prio* (value of priority to make decisions for), *decision* (map from transaction ID to decision for already decided transactions)
Output: Map from transaction ID to decision

1: *prioGrp* $\leftarrow$ Values in *prio* that map to *txnPrio*
2: *higher* $\leftarrow$ Values in *prio* that map to a priority $>$ *txnPrio*
3: Remove rejected transactions from *prioGrp*
4: **for** t $\in$ *prioGrp* **do**
5:     **for** c $\in$ (*conflicts*[t] $\cap$ *higher*) **do**
6:         **if** *decision*[c] = ACCEPT **then**
7:             *decision*|*[t]* $\leftarrow$ REJECT
8:             *prioGrp* $\leftarrow$ *prioGrp* $-$ {t}
9:         **else if** *decision*[c] = DEFER **then**
10:             *decision*[t] $\leftarrow$ DEFER
11:         **end if**
12:     **end for**
13: **end for**
14: **for** t, t′ $\in$ *prioGrp* **do**
15:     **if** t conflicts with t′ **then**
16:         *decision*[t] $\leftarrow$ DEFER
17:         *decision*[t′] $\leftarrow$ DEFER
18:     **end if**
19: **end for**
20: **return** *decision*

Algorithm 3.5 UPDATECONFLICTS(*recno*,*deferred*)
Input: *recon* (reconciliation number), *deferred* (IDs of deferred transactions)

Clear all conflict state (conflict groups, dirty values) from reconciliation *recno*
for $t \in$ *deferred* **do**
  upEx[t] $\leftarrow$ the flattened update extension of t
  Remove from *upEx*[t] all clean updates inapplicable at *recno*
  Mark *upEx*[t] dirty at *recno*
end for
conflicts $\leftarrow$ FINDCONFLICTS(*deferred*,*upEx*)
conflictGroups $\leftarrow \emptyset$
for $t \in$ *deferred*, $t' \in$ *conflicts*[t] **do**
  for conflict $\langle type, value \rangle$ between t and t' **do**
    Add $\{t, t'\}$ to *conflictGroups*[$\langle type, value \rangle$]
  end for
end for
for $\langle type, value \rangle \in$ *conflictGroups.keys* **do**
  Combine compatible txns for $\langle type, value \rangle$ into same option
end for
Record *conflictGroups* as conflict set for *recno*

$O\left(t^2 + tua\right)$.

Proposition 2 (Correctness of RECONCILEUPDATES)**.** *The* RECONCILEUPDATES *procedure given in Algorithm 3.1 satisfies the conditions given in Definition 8.*

Proof. Condition 1 of Definition 8 enforces that decisions from a previous reconciliation are never overruled. It is satisfied, since transaction extensions that contain rejected transactions are rejected at line 4 of the CHECKSTATE method. Otherwise, since RECONCILEUPDATES only considers newly arrived transactions and their transaction extensions, which by definition only contain transactions that have not yet been accepted.

Conditions 2a and 2b are enforced by line 2 for transactions deferred in a previous reconciliation, since any conflicts with or dependency on a deferred transaction will cause the update extension under consideration to touch a dirty value. Conflicts with transactions deferred during the current reconciliation (which will also catch dependent transactions, since they must modify an overlapping set of keys) are caught at line 10 of the DOGROUP method. Condition 2c is caught at line 16 of DOGROUP.

Condition 3a is enforced at line 4 of CHECKSTATE. Condition 3b is checked at line 7 of DOGROUP. By considering the trusted transactions in decreasing order by priority, RECONCILEUPDATES greed-

ily ensures that condition 3b is satisfied; since lower priority transactions can never affect whether higher priority transactions are accepted, the lower priority ones can be considered independently in subsequent iterations. Condition 3c is caught at line 6 of CheckState if the instance was not compatible with the transaction before any transactions were accepted during this reconciliation (if the instance was compatible at the start of the reconciliation but was made incompatible during the reconciliation, then the transaction must also conflict with some transaction of higher priority that was already accepted, and was rejected for violating Condition 3b above).

The remaining conditions are trivially satisfied. The algorithm does not reject or defer transactions that it could accept, no transaction is given more than one decision, and the instance is updated. Lines 14 to 18 of ReconcileUpdates omit the complexity of applying overlapping update extensions correctly, but this is trivial to implement by recomputing upEx[t] immediately before applying it. □

3.4 Update Store

The update store's fundamental role is to archive the transactions published by participants, and to retrieve from storage the transactions that a participant will need during during the reconciliation process. It must also associate a timestamp (the aforementioned *epoch*) with each publication or reconciliation operation, so it can determine which transactions have been published since a participant last reconciled. All other state, such as deferred transactions, conflicts, participant state, trust conditions, and which transactions each participant has accepted or rejected, can at least in theory remain private data to each participant.

Such a system, however, would require a great deal of communication across the network, as each update needed during reconciliation would have to be requested individually. Our implementations, therefore, move the sets of applied and rejected transactions from the participant into the update store, along with the trust conditions; this allows the update store to determine that some transactions need not be retrieved, and thereby reduces that amount of network traffic. An additional result of this approach is that each participant contains only soft state; it is possible to reconstruct the entire state of the participant, up to his or her last reconciliation, from the update store. This does have the side effect of pushing potentially sensitive information into the system, which may be implemented on a central server or using a distributed system. If privacy of this information is of paramount im-

portance, a simpler update store implementation should be used; even if this is done, however, it may still be possible to deduce some sensitive information from the requests made to the store. For the rest of this chapter, and indeed this thesis, we attempt to make as full use as possible of the resources available in a distributed system, and do not focus on privacy or security.

For these reasons, we implement an update store with the following basic operations:

- publish transactions from a participant,
- record that a participant has accepted and rejected certain transactions,
- record that a participant has decided to reconcile and associate with that reconciliation operation a particular set of published transactions,
- retrieve the current reconciliation number of a participant, and
- retrieve all of the transactions (trusted transactions and their unapplied antecedents) that a participant may need to see in order to perform its most recent reconciliation, along with the priorities associated with the fully trusted transactions in that set.

In order to perform these operations efficiently, the update store must log all of the updates published and their epoch, which transactions each participant has accepted or rejected, the current epoch, the epoch corresponding to each participant's previous reconciliation, and the trust conditions for each participant. To facilitate more efficient implementations, the operations are rather high level, allowing significant potential for parallelism and other batch computation; in particular, we chose one operation to retrieve all relevant transactions, rather than operations to, say, get a list of trusted transactions and to retrieve transactions one at a time.

In essence, the update store abstraction hides the details of communication between all participants from the reconciliation algorithm. We now describe at a high level two means of providing the update store functionality. The first uses a standard relational database, and the second is based on the Distributed Hash Table, or DHT (Rowstron and Druschel, 2001; Ratnasamy et al., 2001; Stoica et al., 2001), a peer-to-peer overlay network. The distributed implementation builds directly over an off-the-shelf DHT. Our experience with this implementation, as discussed later in this section, suggested the need for consistent replication and reliability features, which we show how to add in Chapter 4. That chapter shows how to implement a declarative peer-to-peer storage and query

processor that can both serve as a reliable update store and execute the update translation queries generated by update exchange. The more naïve DHT-based approach we show here is much simpler, but lacks the reliability and query processing capabilities we implement for the more general case.

Relational Database Update Store

Relational database technology provides an efficient way to implement a centralized update store. Commercial RDBMSs offer high performance and durability, both important characteristics in a system such as ours, and ease implementation by providing a high-level interface that deals with all synchronization issues automatically. The main potential problem with such an approach is that it requires dedicated hardware and typically some amount of human intervention to tune and maintain. Here we highlight some of the more interesting and innovative aspects of our design; many features are easy to implement, given that the relational database abstraction is high-level and the data being stored is already relational.

In our implementation, an epoch count (implemented using an SQL sequence) is used to timestamp each batch of transactions that it is published. Since publishing is not instantaneous, each participant records when it has started publishing, and also when it has finished. We decouple publishing from reconciliation to support greater concurrency: when a participant requests to reconcile after publishing, it determines the latest epoch not preceded by an "unfinished" epoch, and it uses this as its reconciliation epoch. No additional transactions will be published by any participant prior to this point. The inputs to reconciliation, then, are any transactions whose epoch number lies between the participant's prior reconciliation epoch and this new epoch.

Implementing this approach requires care to avoid sacrificing performance. The series of epoch numbers can contain gaps if reconciliations are rolled back or aborted; therefore each publishing participant must record when it has finished writing all transactions to the database, as mentioned above. However, we also want to allow as many participants as possible to publish updates simultaneously. Repeatable read isolation at the DBMS level prevents race conditions: when the reconciling participant determines the epoch to associate with its reconciliation, it immediately stores that value in the reconciliations table and commits the transaction, releasing all locks. Thus it holds an exclusive lock on the epochs table just long enough to determine the largest stable epoch number; thereafter reconciliation operations are decoupled from the epochs table. By minimizing the time that lock is

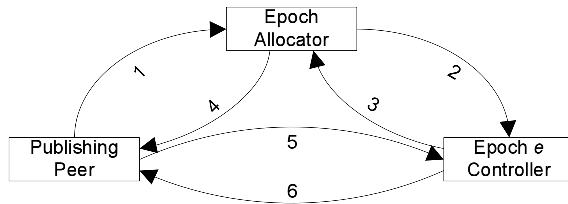
held, we enable maximum concurrency in publishing updates, as well as in the operation of reconciling. As long as no participant is recording its decision to reconcile, there is no limit on the number of participants that can simultaneously start reconciliation, publish updates, or record that they have finished publishing.

Additionally, the richness of SQL makes it easy to express the more complex operations needed for the update store in a succinct manner. Trust condition evaluation can be pushed into an SQL query, filtering out untrusted transactions before they leave the database. Recursive queries, if available, can be used to compute transaction extensions within the database as well. Both techniques can reduce the amount of communication between the reconciling participant and the database, and the load on that participant. extensions are sent over the network.

DHT-Based Store

We have also explored how to create a distributed implementation of the update store. In particular, we explored an update store based on the Distributed Hash Table, a well-studied peer-to-peer overlay network. A peer-to-peer approach has several virtues. First, it is easy to scale, since adding more nodes increases the available storage space and processing power. Second, peer-to-peer networks are self-configuring, so it is possible to create a truly self-managing system, which is important for easy setup of an Orchestra instance. While peer-to-peer networks are often also used for their ability to scale to large numbers of nodes, we do not make use of that property. We use a DHT for its ability to perform self-configuration, including redistributing data as nodes join and leave the system, with no manual intervention. Additionally, the DHT nodes can be less powerful computers, perhaps underused machines owned by some of the participants in the CDSS.

In our DHT-based update store, work (both storage and computation) is spread over the entire network of participants, using transaction identifiers and epochs as keys. The participants store three kinds of data, and each responds to several kinds of messages, which are described below in detail. In this implementation, we assume successful message delivery and postpone a study of fault-tolerance to future work. We do not describe in detail how to incorporate redundancy into the system, to deal with the failure of individual nodes; we rely on replication of all data stored in the system to ensure that the system can continue to operate in the event of node failure. We postpone a more general discussion of reliability to Chapter 4, where we build a more general reliable DHT-based



The messages sent are request epoch (1), begin epoch e (2), confirm epoch begun (3), begin publishing at epoch e (4), publish transaction IDs for epoch e (5), and confirm epoch finished (6). After this the publishing participant can send the transactions for epoch e to their transaction controllers.

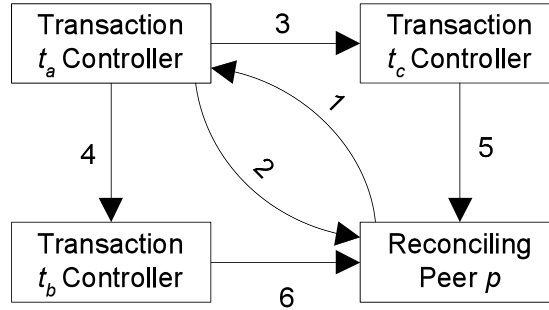
Figure 3.5: Epoch publication in the DHT store

query processor.

One participant, the owner of a predesignated key, keeps track of the epoch count. When a participant wants to publish updates, it requests the next epoch count from this *epoch allocator* (see Figure 3.5). A distributed counter, made reliable through such standard techniques as Paxos (Lamport, 1998) or PBFT (Castro and Liskov, 2002), could also be used to allocate epochs. The epoch allocator informs the *epoch controller* for this epoch (the DHT participant who "owns" the hash value of the epoch) that this participant wants to publish updates, and then returns the epoch count to the requesting participant. After sending the epoch number to the requesting participant, the epoch allocator increments its epoch counter.

The participant then sends each of the transactions it wishes to publish during that epoch to the participant that owns the hash of the ID of that transaction (the *transaction controller*). By taking the hash of the transaction ID and storing the transaction at the node that owns that key, we are distributing the largest piece of data (the update log) across all participants approximately evenly. After a participant publishes its set of transaction, it transmits their IDs to the epoch controller, which records the list of transactions associated with that epoch. The epoch controller concludes by marking the epoch as complete. This sequence of operations (updating the epoch allocator, then the epoch controller, then publishing the transactions, and finally marking the epoch controller's record as finished) is necessary to avoid race conditions when determining the most recent "stable" epoch, the last epoch for which there are no publication operations in progress, which must be determined during reconciliation.

When a participant reconciles, it first needs to associate an epoch with the reconciliation operation, to determine which transactions it will consider. It requests the most recent epoch from the



In this example $\mathtt{p}$ requests reconciliation information for transaction $\mathtt{t_a}$. $\mathtt{p}$ has already applied $\mathtt{t_b}$, but has not decided $\mathtt{t_a}$ or $\mathtt{t_c}$. $\mathtt{t_b}$ and $\mathtt{t_c}$ are antecedents of $\mathtt{t_a}$; $\mathtt{t_b}$ has other antecedents, but $\mathtt{t_c}$ has none. The messages sent are request $\mathtt{t_a}$ (1), send $\mathtt{t_a}$ (2), request $\mathtt{t_c}$ for $\mathtt{p}$ (3), request $\mathtt{t_b}$ for $\mathtt{p}$ (4), send $\mathtt{t_c}$ (5), and $\mathtt{t_b}$ not relevant (6).

Figure 3.6: Example transaction retrieval in the DHT store

epoch allocator, and uses that information to request the contents of all epochs since its last reconciliation from their respective epoch controllers. It uses this information to determine the most recent stable epoch, and records this as its reconciliation epoch at its *participant coordinator*. Then, for each epoch since the participant's prior reconciliation, it requests the set of transactions published in that epoch from the epoch controller, and then requests that set from the transaction controllers. Each transaction controller either sends back the requested transaction, its priority, and a set of antecedents, or a notification that the transaction is untrusted or irrelevant (due to having already been accepted or rejected). The reconciling participant maintains a pending transactions set, to which it adds antecedents and from which it removes received (or irrelevant) transactions. This procedure is visualized in Figure 3.6; it minimizes the the number of hops across the DHT that must occur before all necessary transactions have been received. The recursive portion transaction dependency processing happens in a distributed fashion, instead of requiring the reconciling node to actively request each antecedent transaction *after* receiving a dependent transaction. When the pending transactions set becomes empty, the participant runs the centralized (client-centric) reconciliation algorithm described in Section 3.3. That algorithm notifies the appropriate transaction controllers when a participant decides to accept or reject transactions.

When the transaction controller receives a request for a transaction during reconciliation, it first determines the priority with which the reconciling participant trusts that transaction; it can do that since the trust conditions are globally available in this approach. If the participant has previously

accepted or rejected this transaction, that information is stored locally, since it is owned by the transaction controller. If the transaction is trusted and not already decided, the transaction controller sends the transaction and the priority back; if not, it sends a message saying the transaction is not relevant. It also sends a message requesting the antecedent for reconciling participant for each antecedent transaction. When a transaction controller receives such a requesting antecedent message for a particular participant, it checks to see if the participant has accepted or rejected that transaction (which again is stored locally). If it has, it sends a message to the reconciling participant saying that the transaction is not relevant for the reconciliation; otherwise, it sends the contents of that transaction back to the participant, and sends a requesting antecedent message for that participant for each antecedent transaction.

The approach described above is free of race conditions due to delays in message propagation across the DHT, and since we assume reliable message delivery, it will function correctly if the set of nodes participating in the DHT holds constant. Background replication can be used to ensure that state moves between nodes in the DHT as needed; however, since the data for many keys is mutable, background replication combined with churn might lead to reliability issues. This was the motivation for the more general work on reliable storage we describe in Chapter 4.

Additionally, in the process of developing an implementation of the DHT-based update store, we learned the extreme importance of data placement for good processing performance. We observed that, for many workloads, network operations become the dominant cost, and should be avoided as much as possible. While this is not in itself surprising, we were very surprised by the effect that optimizing based on this idea had on performance in preliminary experiments. This observation is the reason that records of which transactions have been accepted or rejected are stored with the transactions, eliminating an extra network hop when performing the recursive transaction retrieval. Similarly, the transaction retrieval is performed in a distributed fashion, with the node that holds each transaction requesting its antecedent transactions, rather than the reconciling node, since this reduces the overall latency of retrieving a chain of dependent transactions, and lessens the load on the reconciling participant. We will explore ways to generalize data colocation and distributed many-to-many communication to improve performance and reduce latency in the context of reliable, declarative query processing, also in Chapter 4.

3.5 Experimental Analysis

We have implemented the reconciliation algorithm of Section 3.3 above in Java, and also constructed a centralized update store, built in Java over a major commercial RDBMS, and a distributed store, based on FreePastry (Rowstron and Druschel, 2001). In the analysis of our prototype presented here, we explore configurations of up to fifty participants, which we feel is representative of medium-sized ORCHESTRA instances.

Given that no comprehensive workload already exists for bioinformatics data sharing, we developed a synthetic workload generator based on the Swiss-Prot bioinformatics database, which contains organisms, proteins, and protein functions. The simulator mimics the process of incrementally maintaining a curated database like Swiss-Prot. Each transaction consists of a series of insertions or replacements over the Function relation, and if it is an insertion, a secondary table of database cross-references is updated to include a reference for the new key. On average, 7.3 such tuples are inserted into the secondary table for each value inserted into the primary table; these are taken directly from Swiss-Prot. The key of the Function table to update is chosen from the approximately 280,000 proteins present in Swiss-Prot at the type of these experiments. The key to insert or update is chosen according to a heavy-tailed Zipfian distribution with characteristic $s = 1.5$, thus simulating the "hotspots" of contention or current research that are likely. The function to associate with that key (either through an insertion or an update) is chosen from the approximately unique 7,000 functions present in Swiss-Prot, weighted by their frequency in the database; this distribution also has Zipfian characteristics. This definition means that conflicts can only arise on insertions to or modifications of the Function relation.

We feel that transactions are likely to be small, since each transaction likely represents the determination of a single research paper; therefore transactions affecting a few proteins are likely, but not transactions affecting hundreds or thousands. The simulation is turn-based: it cycles through the list of participants, and for each participant performs an action (insert a new transaction, publish and then reconcile, randomly resolve a previously recorded conflict, or do nothing). We change the probabilities of these various actions to alter the simulation parameters, such as the average number of transactions published between reconciliations. All participants in the simulation trust each other at the same priority for all updates, which means that *all* conflicts must be manually rather than automatically resolved. It also maximizes the work of the reconciliation algorithm, since all transactions

retrieved during a reconciliation must be checked for conflicts, rather than just those with the same priority. Recall that querying a participant's customized local instance requires no work on behalf of the CDSS, since it was synchronized during the reconciliations; therefore these operations, which may in fact be the most frequent, are not in our simulation.

Our experimental evaluation focuses on two metrics. The first metric is, as one might expect, execution time of the reconciliation procedure. We wanted to understand the effects of changing various system parameters on execution time, and to validate that our prototype implementation performs as expected. Clearly any data sharing system needs to continue to perform well as we increase the number of participants in the system and the amount of data in the system. We additionally wanted to explore the relative costs of various operations in our implementation of ORCHESTRA. In particular, we wanted to know if dominant costs of reconciliation were in transaction retrieval from the update store or in the client-centric reconciliation algorithm. Clearly this would guide our decisions as to where work on improving performance should be focused.

The second metric does not focus on validating our ORCHESTRA implementation, but rather on the CDSS model itself. It is what we term the *state ratio*, the average number of values in all participants' states for a key (including lack of a value). It measures the amount of divergence between databases, ranging from one, if all the participants have exactly the same state, to the number of participants, if there is no overlap at all between the participants' states. Since a lower ratio indicates more of the data is shared, we consider a smaller value for this metric to generally indicate higher quality result. These experiments were performed to answer questions posed by anonymous reviewers of Taylor and Ives (2006). They were concerned that an ORCHESTRA instance could quickly degenerate into a situation where virtually no data was shared between participants due to conflicts, especially between antecedent transactions. Clearly from an implementation perspective, a higher state ratio is not bad, since the system is still performing according to the rules outlines in Definition 8; however, complete divergence of participants' instances in a simulation would raise questions about the appropriateness of the model. It would have been preferable, of course, to compare against another system for collaborative data sharing, but none exists; we therefore use the state ratio as our benchmark of result quality.

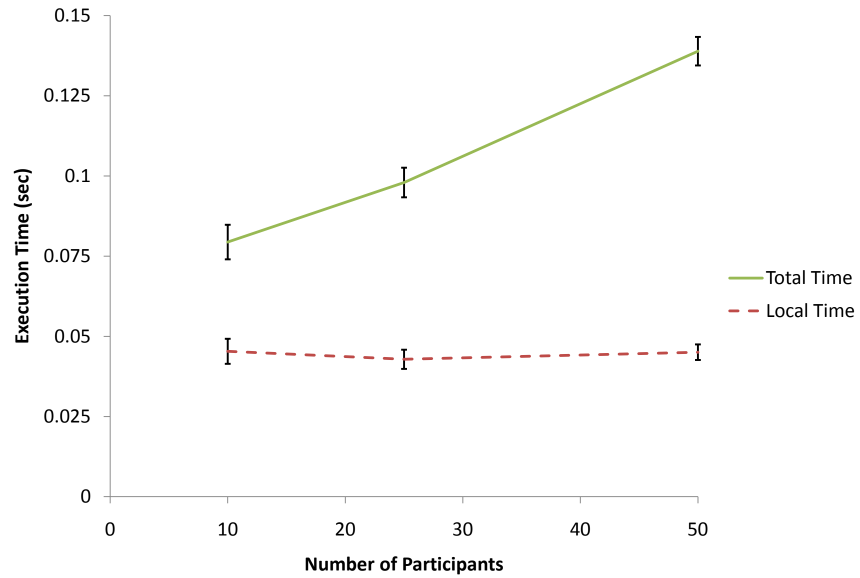
Our experiments examine the effects of transaction size, number of participants, and reconciliation frequency on these metrics. Experiments were run at least five times, and the mean of these trials is presented; 95% confidence intervals, calculated using the the standard deviation of a mean,

are given in all figures. Unless indicated, all experiments were performed with ten participants. All of the running times quoted in this chapter are the averaged over all trials over all participants. For the experiments using the central store, the RDBMS ran on a dual Xeon 3.0GHz server with two gigabytes of RAM running Windows Server 2003, and the client ran on a 2.8GHz Pentium 4 with one gigabyte of RAM running SuSE Linux 9.2. The computers were connected via switched 100Mb Ethernet. For experiments with the distributed store, all nodes were run on the Windows server, with a delay of at least 500 microseconds added to every message (and reply) transmission; this is a reasonable delay within a physical building and should allow direct comparison to the central store performance. All machines used Sun's JRE 1.5.0. We feel that this rather modest configuration represents a worst-case scenario for the resources available to the system. An actual Orchestra instance would likely contain more powerful nodes, and would certainly achieve at least some benefit from the parallelism that comes with an increased number of nodes; our experimental results only show the overhead.

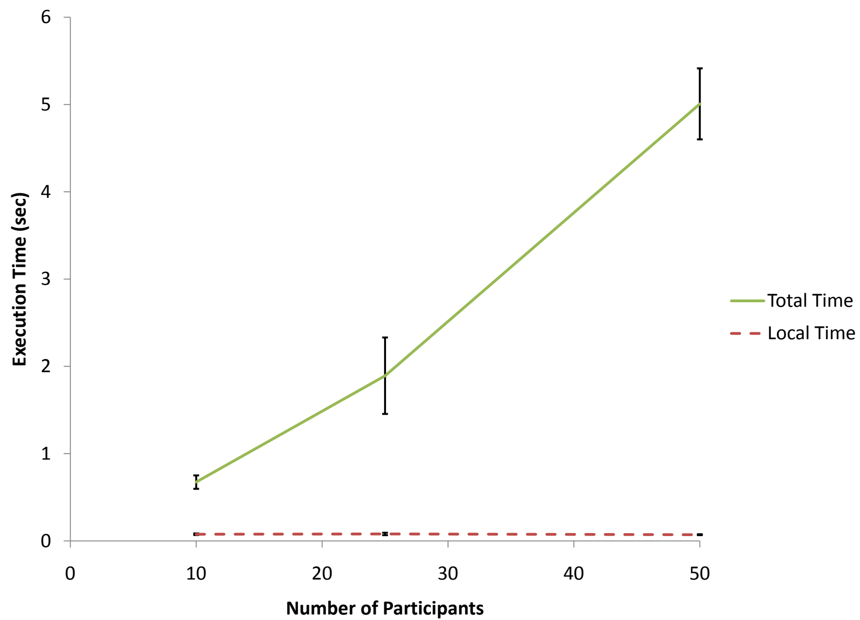
Exploration of Running Time

We began our study of running time with an obvious scaling experiment. With each participant publishing an average of 20 transactions between reconciliations and performing an average of 10 publish/reconciliation operations, we explored the the effect of increasing the number of participants on reconciliation time; each transaction contains only one update (to the Function table) that may conflict with other data or transactions. The results of this experiment are given in Figure 3.7, which shows that total reconciliation time (per participant) depends linearly on the number of participants, and therefore on the amount of data being considered; this is true for both reconciliation over the centralized update store and the distributed update store. More interesting, in the case of the results for the distributed update store, shown in Figure 3.7b, the cost of retrieving the updates totally dominates the cost of execution the reconciliation algorithm. This indicates to us that we should focus on improving the performance of the distributed update store, rather than the performance of the reconciliation algorithm. We will return to this challenge in the work on distributed query processing in Chapter 4, which can be used to implement an efficient, reliable update store in a declarative fashion.

We were also concerned about performance if reconciliation is very frequent or relatively infrequent. In very frequent reconciliation, fixed per-operation costs might dominate. In very infrequent

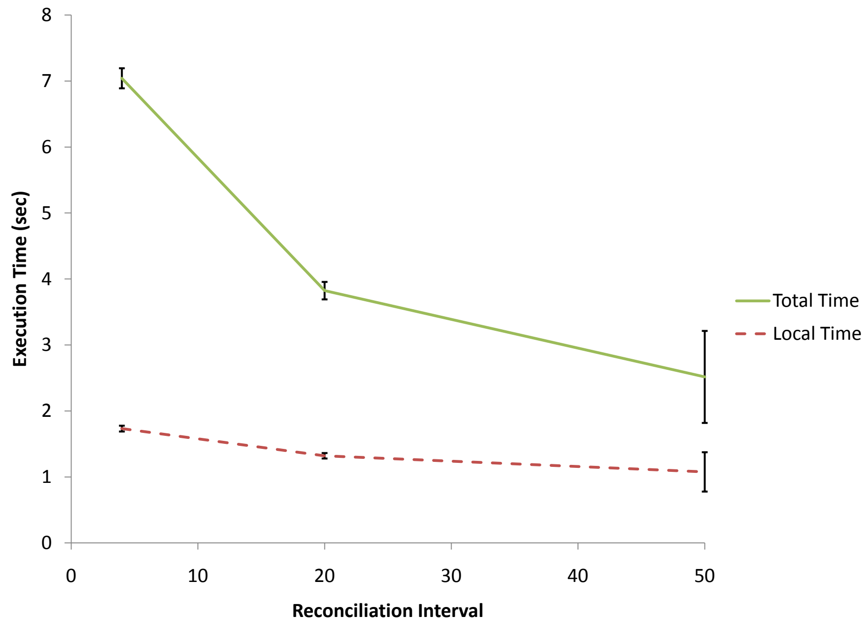


(a) Centralized (RDBMS) Update Store

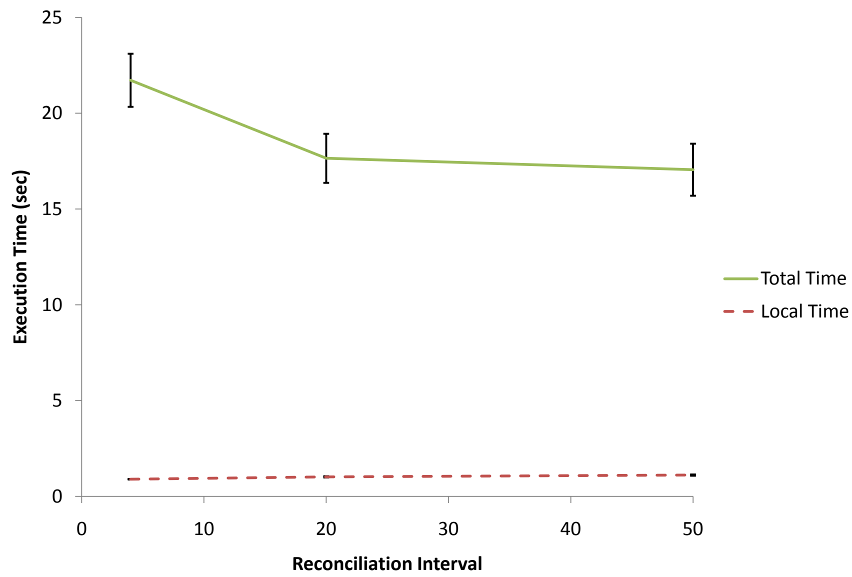


(b) Distributed (Peer-to-Peer) Update Store

Figure 3.7: Effect of number of participants on execution time. Local time is the time to execute the client-centric reconciliation algorithm, while total time also includes the time to retrieve the necessary transactions from the update store.



(a) Centralized (RDBMS) Update Store



(b) Distributed (Peer-to-Peer) Update Store

Figure 3.8: The effect on execution time of varying reconciliation interval (the average number of transactions performed by each participant between its reconciliations) while holding transaction size at one.

reconciliation, the larger sets of potentially conflicting transactions may cause performance problems. Figure 3.8 shows the average execution times participant when each participant publishes approximately 500 transactions, but performs a publish/reconcile operation after a varying number of these transactions; we term the average number of transactions performed between each publish/reconciliation operation to be the *reconciliation interval*. Again, we show results for both centralized and distributed update stores, and break out time spent in the client-centric reconciliation algorithm, from the time spent retrieving the necessary transactions. As one would expect, the time needed to perform the client-centric reconciliation algorithm (the local time) becomes more varied as the reconciliation interval increases, due to the possibility of more conflicting transactions; however, the mean is not strongly correlated with the reconciliation interval. Overall, therefore, reconciliation interval does not have a strong connection to the total amount of time needed to choose consistent subsets of transactions to apply. Of course, the results here show the average per-participant execution time for all of its reconciliations; the time per-reconciliation will change.

Most of the variation in the performance shown in Figure 3.8 comes from the cost of retrieving the updates from the update store. The centralized update store results shown in Figure 3.8a show that in that case there is a large fixed per-reconciliation cost to retrieve the needed transactions from the update store; this is not surprising, giving the relatively high fixed cost of posing any query to a RDBMS. For the distributed update store shown in Figure 3.8b, where the requests to follow antecedent transaction chains dominate the running time, the penalty for more smaller reconciliation intervals is much less pronounced. In both cases, however, Orchestra users would have the flexibility to reconcile frequents or more infrequently without having a dramatic impact on overall system performance. Once again, we observe that retrieving transactions from the distributed update store is a dominant cost; by focusing on retrieval and query processing performance in Chapter 4, we can reduce this cost in a distributed update store build over our peer-to-peer data storage and processing layer.

Exploration of the State Ratio

As mentioned previously, we were also concerned that an Orchestra instance could devolve into a collection of databases with almost no overlap between them, due to conflicts and transactional dependencies. We therefore explored the scenarios from the previous section from the perspective

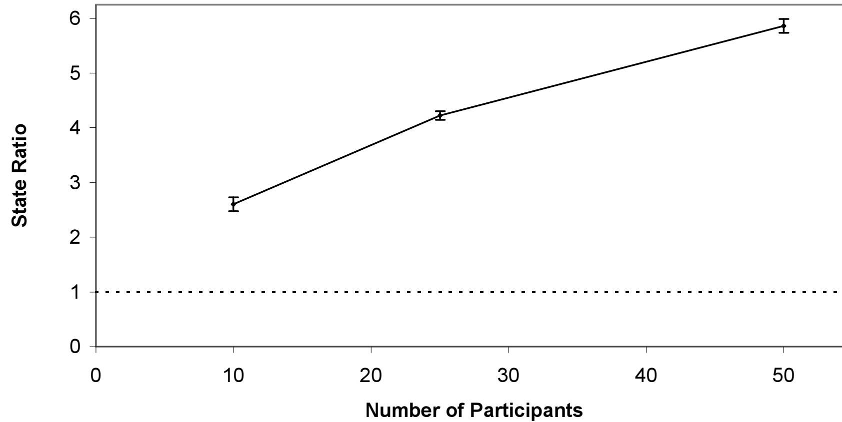


Figure 3.9: Effect of number of participants on state ratio. Note that the relationship, at least for the skewed distributions studied here, is sublinear.

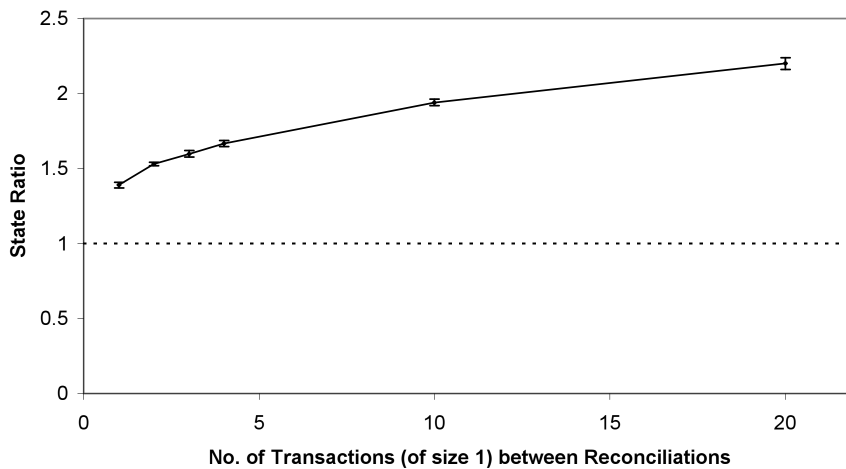


Figure 3.10: The effect on state ratio of varying reconciliation interval

of result quality, as measured by the state ratio; recall that this ranges from one to the number of participants, with a higher number indicating greater divergence between participant instances.

Figure 3.9 shows the effect on state ratio of increasing the number of participants while keeping the reconciliation interval and average number of reconciliations for each participant constant. Once again, all transactions contained only one update. Increasing the number of participants (and the number of data) does increase the mount of divergence, but the effect seems to be rather sublinear; this bodes well for the amount of overlap possible in larger Orchestra instances.

Figure 3.10 repeats the second experiment from the previous section, again looking at state ratio instead of running time. Here we see the effects increasing the reconciliation interval while keeping

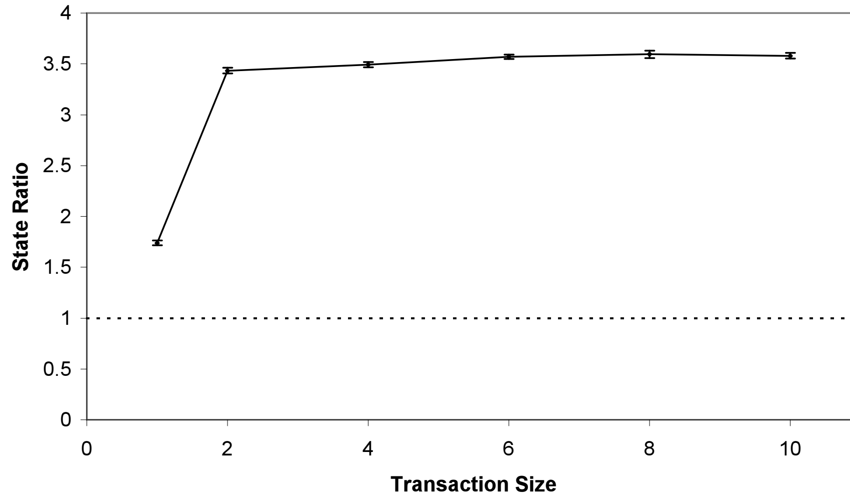


Figure 3.11: The effect of varying transaction size on state ratio, while holding the number of updates between reconciliations constant.

the average number of single-update transactions performed by each of the 10 participants fixed at 500. As in the previous experiment, the results are promising. Increasing the reconciliation interval results in an increase in the state ratio, as one would expect, but once again the effect is decidedly sub-linear; this indicates that it should be possible to reconcile much less frequently without increasing the amount of divergence too much.

Figure 3.11 examines the relationship between transaction size and the overall amount of data shared between participants. Once again, each of the ten participants performed approximately 500 updates to the main protein function table; they were grouping into transactions of varying sizes. As one might expect, increasing transaction size increases the state ratio: larger transactions increase the number of transactions that conflict with each other, thus result in more overall (inconsistent) state within the CDSS. Surprisingly, while going from single-update transactions to transactions with two updates greatly increases the state ratio, further increases in transaction size have negligible effect, at least for our synthetic workloads. We believe that this is because the number of direct and indirect antecedents of a transaction is exponentially related to the number of updates in that transaction, since each update may depend on an independent antecedent. Therefore, if update chains of length n are possible, and each transaction has m updates in it, a transaction may depend on up to m^n other transactions. If, as in our workload, there is a relatively small set of "hot" items that are likely to cause conflicts, a transaction chain will likely be rejected if it contains any of those hot items, as its unlikely

that the transaction chain's value matches the participant's current state. The probability that one of the m^n antecedent transactions conflicts with a hot item increases much more rapidly when m is small; once m is large it is already very likely that one of the antecedents will conflict with a hot value.

In summary, the experiments on the state ratio show that result quality is not affected greatly by reconciliation interval or the number of participants. It is more affected by transaction size, though the effects do not seem much more pronounced for large transactions than for medium-sized ones. However, further exploration of this topic, based on real data instead of synthetic workloads, is necessary before any definite conclusions can be reached. The effects we saw here do not necessarily hold for other workloads. Performing the user studies needed to get actual workloads would be an interesting challenge in and of itself, but is well beyond the scope of this document, and the expertise of its author.

3.6 Conclusions and Analysis

In this chapter, we detailed the semantics of reconciliation. We showed how trust conditions and priorities determine which transactions are accepted, and detailed efficient algorithms for determining a maximal set of compatible transactions. We explored the effects of various parameters on ORCHESTRA system performance, both in terms of execution time and in terms of the amount of divergence that reconciliation allows to remain in the system. We showed that our reconciliation procedures can operate efficiently over data stored both in a single-server RDBMS and in a peer-to-peer storage and retrieval layer built over an off-the-shelf distributed hash table.

The initial peer-to-peer storage layer validated that efficient storage and retrieval of the participant update logs was possible, and showed the extreme importance of data locality in this setting. However, it lacked reliability features, and could return incorrect or inconsistent results under churn. While it could distribute the work of storing and retrieving updates, it could not distribute the execution of update exchange queries, needed for the complete ORCHESTRA case of multiple schemas. In the next chapter, we describe a declarative peer-to-peer storage and query processing layer that adds support for complex SQL queries, and guarantees correct results under node failure and churn. The reconciliation procedures described here can be used in concert with an update store, implemented in a declarative way, over this storage and query processing layer, to provide a fully functional and reliable implementation of ORCHESTRA over an easily setup and scalable peer-to-peer system.

Chapter 4

Reliable, Declarative Peer-to-Peer Storage and Computation

In this chapter, we address a key limitation in the techniques presented in Chapter 3. That chapter described how to reconcile conflicting updates from other participants, and in Section 3.4 proposed several methods for implementing the *update store*, the global update log that is the principal shared state in ORCHESTRA. Recall that the update log contains a permanent record of all updates published to the system. Since all updates must be available at all times for a participant to be able to reconcile, the system cannot simply rely on each participant storing the definitive record of its own updates; they are instead published to an archive that ensures high availability.

Both of the proposed implementations of an update store have significant shortcomings. The approach of using a centralized relational database is clearly not scalable, requires significant computer resources and human support to maintain, and is a single point of failure. The distributed hash table-based approach was promising, in that it distributed storage of the update log in a scalable way, required little management and virtually no setup, and did not have a single point of failure. It did not, however, make the same strong consistency guarantees as the relational database approach, which are necessary for correct ORCHESTRA system operation, and its hard-coded execution mechanisms made it inflexible to change to add features or improve performance. Additionally, it does not integrate with the update translation and provenance work from Chapter 2, which executes over an SQL database. In this chapter, we show how we can provide a unified substrate which can fulfill the storage and query processing needs of both of these ORCHESTRA components by building a highly scalable

and reliable versioned storage and query processing system for ORCHESTRA. Since it provides relational storage and execution of SQL queries, it can be used as the execution layer for both the update store and the update translation services. To avoid the need for dedicated servers, we employ an *ad hoc* cloud of nodes, consisting of the existing CDSS nodes, possibly supplemented on an as-needed basis with machines leased as-needed from cloud services such as Amazon EC2; since our system is entirely self-configuring, adding or removing nodes from it while running is supported (and indeed expected). By using a peer-to-peer approach, we avoid the single point of failure of a central relational database, gain easy scalability, and avoid the need for extensive human management of the system.

This chapter in structured as follows:

- Section 4.1 gives the specific goals of our peer-to-peer storage and query processing substrate, which focus on performance and reliability over scalability.

- Section 4.2 details our modifications to the standard data partitioning techniques used in traditional distributed hash tables, such as Pastry (Rowstron and Druschel, 2001). We customize them for a smaller, more stable environment, and provide greater transparency of operation to the layers above.

- Section 4.3 describes our novel indexing technique that allows reliable access to *versioned* data, including efficient processing of updates, in the presence of network churn and replication lag.

- Section 4.4 describes our partitioned-parallel distributed query processor, including new operator algorithms that use the versioned indices. It also shows how we detect and compensate for node failure by either completely restarting or *incrementally* recomputing the query, while ensuring the correct and complete query results are returned.

- Section 4.5 presents the results of experiments, using a standard OLAP benchmark and schema mapping tasks, that validate the performance of our techniques across local and cloud computing nodes. We also validate our methods under different network settings, and induce failures to study the benefits of incremental recovery.

Our goal is to provide the benefits of peer-to-peer architectures like Pastry (Rowstron and Druschel, 2001), PIER (Huebsch et al., 2005; Loo et al., 2004), Seaweed (Narayanan et al., 2008), CAN (Ratnasamy et al., 2001), and CHORD (Stoica et al., 2001), which typically offer support for

autonomous domains with no common filesystem, transparent handling of membership changes, and plug-and-play operation. We *hybridize* this with the benefits commonly associated with traditional parallel DBMSs and with emerging cloud data management platforms, like Google's GFS (Ghemawat et al., 2003) and Bigtable (Chang et al., 2008), Amazon's S3 (DeCandia et al., 2007), and Yahoo!'s Pig (Olston et al., 2008) and PNUTS (Cooper et al., 2008), which offer efficient data partitioning, automatic failover and partial recomputation, and guarantees of complete answers. By exploiting some of the features of the CDSS domain (namely smaller numbers of nodes, lower churn, and batch-oriented publication of updates), we can avoid what we perceive to be the negative aspects of each architecture: the lack of completeness or consistency guarantees in peer-to-peer query systems, and requirements for shared filesystems and centralized administration in the aforementioned cloud data management services.

In particular, we exploit the fact that our system does not need *all* of the properties provided by existing distributed substrates. Our problem space is less prone to "churn" than a traditional peer-to-peer system like a distributed hash table: we assume that membership in a CDSS, while not completely stable, consists of perhaps *dozens to hundreds* of participants at academic institutions or corporations, with *good bandwidth* and relatively *stable machines*. We support archived storage of data under a *batch-oriented* update load: in a CDSS, users first make updates only to their local storage, and they occasionally *publish* a log of these updates (which are primarily insertions of new data items) to the CDSS. Then they perform an *import* (transforming and importing others' newly published data to their local replica). Only in this step is information actually shared across users, and it is then that conflict resolution is performed. Hence, we *do not need special support for global consistency*, such as distributed locking or version vectors, at the distributed storage level.

We address these needs through a custom data partitioning and storage layer, as well as a new distributed query processor. We develop novel techniques for ensuring versioning, consistency, and failure recovery in order to guarantee complete answers. While we implement and evaluate our techniques within the ORCHESTRA collaborative data sharing system, the techniques are broadly applicable across a variety of emerging data management applications, such as distributed version control, data exchange, and data warehousing.

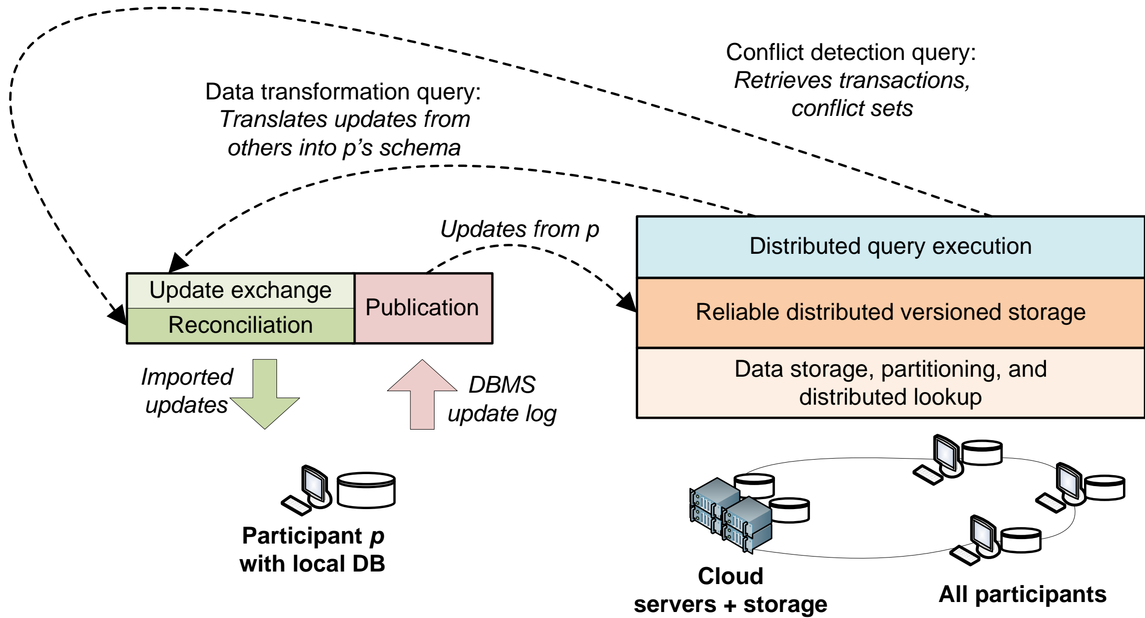


Figure 4.1: Basic architecture components of Orchestra. Chapters 2 and 3 described the components on the left, which execute over a storage and data processing system. In this chapter, we focus on implementation of the Orchestra reliable, peer-to-peer storage system and query processing engine.

4.1 System Architecture and Requirements

Figure 4.1 repeats Figure 2.2 from Section 2.2, which gave an outline of the architecture of Orchestra. In Chapters 2 and 3, we have focused on the left half of this figure: the logic needed to create and use the SQL queries supporting update exchange and reconciliation, and the modules to "hook" into the DBMS to obtain update logs. In this chapter, we focus on the right half of the diagram: how to implement distributed, versioned storage and distributed query execution. We would like our system to be as "plug and play" as possible, requiring much less administration and tuning than a traditional relational database. We would also like to make use of an *ad hoc* "cloud" of participant nodes, to avoid a central server and point of failure, while maintaining high reliability. In addition, we are particularly concerned with performance in support of update exchange (data transformation) queries, which were by far the main bottleneck in performance in Green et al. (2007a). We also develop capabilities in the query execution layer to support mapping and OLAP-style queries directly over the distributed, versioned data. Data is primarily stored and replicated among the various participants' nodes. However, as greater resources, particularly in terms of CPU, are required, participants may

purchase cycles on a cloud computing service capable of running arbitrary code, such as Amazon's EC2 (considered here) or Microsoft's Azure. Given the automatic failover and replication present in system we describe here, it is trivial to add supplemental nodes when they are needed, and to remove them later if they are not.

In the remainder of this section, we explain the unique requirements of ORCHESTRA and why they require new solutions beyond the existing state of the art. In subsequent sections, we describe our actual solutions.

Data Storage, Partitioning, and Distributed Lookup

As discussed previously, we assume that the participants number in the dozens to hundreds, are usually connected, and together have enough storage capacity to maintain a log of all data versions. Our target domain differs from conventional peer-to-peer systems, where connectivity is highly unstable. We only expect low "churn" (nodes joining and leaving the system) rates, perhaps as participants go down for maintenance or are replaced with new machines. We expect failures to be infrequent enough that keeping a few replicas of every data item is sufficient. We avoid single points of failure, as we want the service to remain available at all times, even if some nodes go down for maintenance.

In a distributed implementation of a CDSS, we need a means of (1) partitioning the stored data (such that it is distributed reasonably uniformly across the nodes), (2) ensuring efficient repartitioning when nodes join and leave, (3) supporting distributed query computation, and (4) supporting background replication. There are two main schemes for doing this in a distributed system: distributing data *page-by-page* in a *distributed filesystem*, and then using a sort- or hash-based scheme to combine and process the data; and distributing data *tuple-by-tuple* according to a key, and using a distributed hash scheme to route messages to nodes in a network. Google's MapReduce and GFS, as well as Hadoop and HDFS, use the former model. Distributed hash tables (Rowstron and Druschel, 2001; Ratnasamy et al., 2001; Stoica et al., 2001) and directory-based schemes use the latter.

Cloud-oriented distributed filesystems like GFS and HDFS suffer from several drawbacks as the basis of a query engine. First, they require a single administrative domain and a single coordinator node (the NameNode), which introduces a single point of failure. Moreover, they use two different distribution models. Base data is partitioned on a page-by-page basis, but can only be accessed directly when being scanned. All multi-pass query operations (joins, aggregation, nesting) must re-

distribute data between nodes in some manner. This is typically done using a MapReduce scheme that partitions the data on keys (mapping them to nodes via sorting or hashing), and then produces joined or aggregated output (in the reduction phase); examples from the literature include Thusoo et al. (2010) and Abouzeid et al. (2009). Since the data must be redistributed for any processing to take place, there is no way to exploit data locality by colocating multiple relations in a way to facilitate common queries.

We instead adopt a tuple-by-tuple hash-based distribution scheme for routing messages: this is commonly referred to as a *content addressable overlay network* and is exemplified by the DHT. Our goal is to provide good performance for the complex queries that arise from update exchange and to tolerate nodes joining or failing, but we do not require scalability to millions of nodes as with the DHT. In Section 4.2 we adapt some of the key ideas of the DHT in order to accomplish this. The unified distribution model affords several benefits. In addition to simplicity, it allows relations to be colocated; relations likely to be joined might be partitioned on the same hash key, reducing the cost of joining them. Alternatively, if one of them is already partitioned on the join key, only the second relation might need to be repartitioned to facilitate the join. This is in contrast to the GFS/MapReduce approach described above, where virtually any operation will involve repartitioning (i.e. mapping) all of the data.

Versioned Storage

Each time a participant in ORCHESTRA publishes its updates, we create a new *version* of that participant's update log (stored as tables). This also results in a new version of the global state published to the CDSS. Now, when a participant in ORCHESTRA imports data via update exchange and reconciliation, it expects to receive a *consistent*, *complete* set of answers according to some version of that global state. We support this with a storage scheme (described in Section 4.3) that tracks state across versions, and manages replication and failover when node membership changes, such that queries receive a "snapshot" of the data according to a version.

When data is stored in a traditional content-addressable network, background replication methods ensure that all data *eventually* is replicated, and gets placed where it belongs when a node fails — but if the set of participants is changing then data may temporarily be missed during query processing. Furthermore, such systems also require the data assigned to each key to be immutable. Similarly,

existing distributed filesystems like GFS and HDFS assume data is within immutable files, and they are additionally restricted to a single administrative domain.

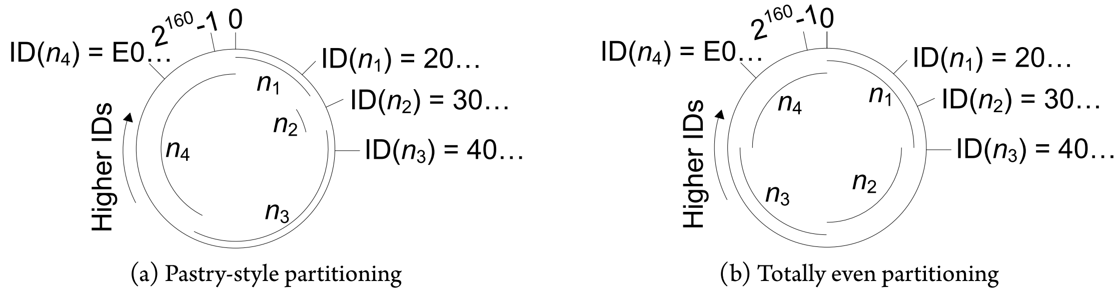
Hence our versioned storage scheme must provide bookkeeping than a traditional distributed hash table, but offers more autonomy and flexibility than a distributed filesystem. In Section 4.2 we describe our customized data storage, partitioning, and distributed lookup layer.

Query Processing Layer

As is further discussed in Section 6.3, a number of existing query processing systems, including PIER (Huebsch et al., 2005; Loo et al., 2004) and Seaweed (Narayanan et al., 2008), have employed DHTs to perform large-scale, "best-effort" query processing of streaming data. In essence, the DHT is treated like a very large parallel DBMS, where hashing is used as the basis of intra-operator parallelism. Immutable data can be stored at every peer, accessed by hashing its index key. Operations like joins can be performed by hashing both relations according to their join key, co-locating the relations to be joined at the same node. Such work has two related shortcomings for our context: multiple data versions are not supported, and their "best-effort" consistency model in the presence of failures or node membership changes is insufficient.

A primary goal is to support efficient distributed computation of query answers; in our case, this is the retrieval of newly available updates and the translation between schemas that occurs during update exchange. The DHT nicely distributes the work of retrieving data from persistent storage and the computation of query results, both of which are potentially the limiting factors in query performance. However, we also need to detect and recover from node failure during query execution. We emphasize that this is different from recovery in a transactional sense: here our goal is to *compensate for missing answers in a query*, ideally without redoing the entire query from scratch (whereas transactional recovery typically does involve aborting and recomputing from the beginning). Failure recovery in query answering requires us (in Section 4.4) to develop techniques to track the processing of query state, all the way from the initial versioned index and storage layer, through the various query operators, to the final output.

Furthermore, we develop techniques for *incrementally* recomputing only those results that a failed node was responsible for producing. Given that every operator in the query plan may be executed in parallel across all nodes, the failure of a single node affects intermediate state at all levels of the plan.



(a) Pastry-style partitioning (b) Totally even partitioning

Figure 4.2: Schemes for partitioning the DHT key space among the participants

Our goal is to restart the query only over the affected portions of the data, and yet to ensure that the query does not produce duplicate or incorrect answers.

4.2 Hashing-Based Substrate

Any scalable substrate for data storage in a peer-to-peer setting needs to adopt techniques for (1) data partitioning, (2) data retrieval, and (3) handling node membership changes, including failures. We describe how our custom hashing-based storage layer addresses these issues, in a way that is fully decentralized and supports multiple administrative domains.

Data Partitioning

Like most content-addressable overlay networks, we adopt a hash-based system for data placement. Similar to previous well-known *distributed hash tables* (DHTs) such as Pastry (Rowstron and Druschel, 2001), we use as our key space 160-bit unsigned integers, matching the output of the SHA-1 cryptographic hash function. It is convenient to visualize the key space as a ring of values, starting at 0 and increasing clockwise until they get to $2^{160} - 1$, at which point they overflow back to 0. Figure 4.2 shows two examples of this ring that we will discuss in more detail.

Most overlay networks assign a position in the ring to each node according to a SHA-1 hash of the node's IP address (forming a DHT ID). Values are placed at nodes according to the relationship with their hash keys. In Chord, keys are placed at the node whose hashed IP address lies *ahead* of them on the ring; in Pastry the keys are placed at the node with *nearest* hash value. The Pastry scheme of partitioning the key space among the participant nodes is shown in Figure 4.2a. Both of these approaches can determine the range a node "owns" given its ID and the IDs of its neighbors. These

schemes are optimized for settings with large numbers of nodes, and assume the nodes will be more or less uniformly distributed across the ring. Each node maintains information about the position of a limited number of its neighbors, as it has a routing table with a number of entries logarithmic in the membership of the DHT. DHTs, as presented so far, often have highly nonuniform (approximately Poisson) distributions of values among the peers. Indeed, in the figure, nodes n_3 and n_4 are together responsible for more than ¾ of the key space, while node n_2 is only responsible for 1⁄16 of it.

The canonical approach to load imbalance in DHTs is to use many *virtual nodes*, also referred to as *virtual servers* in, e.g., Dabek et al. (2001), at each physical node. This makes it much more likely that each physical node receives an approximately even fraction of the key space; even though the amount assigned to individual virtual nodes will vary, the total amount assigned to a large number of them is very consistent with high probability. Virtual nodes have some drawbacks, and in particular, they lead to key space fragmentation. For our purposes, it is very advantageous to assign a single contiguous key range to each node; in addition to reducing the size of the routing table, this improves data retrieval performance by allowing us to colocate data at nearby DHT keys, as discussed in Section 4.3.

For this chapter, we adopt a simple solution we term *totally even partitioning*. We divide the key space into evenly sized sequential ranges, one for each node, and assign the ranges in order to the nodes, sorted by their hash ID. Such an assignment for the same network we examined for Pastry-style partitioning is shown in Figure 4.2b; it distributes the key space, and therefore the data, uniformly among the nodes. In response to node arrival or failure, we redistribute the ranges over the new node set. This approach is more sensitive to churn than the Pastry approach. Pastry has what we term *partitioning resiliency*: only a *neighboring* node's arrival or departure causes a change to a node's owned range. In totally even partitioning, *any* node's arrival or departure will cause a change to a node's owned range. Partitioning resiliency reduces the effect of churn by causing less of the key space (and therefore less data) to be moved when a node arrives or departs[1]. However, in a smaller, low-churn network the performance benefits afforded by more even data distribution outweigh the lack of partitioning resiliency. In this chapter, we our experiments exclusively consider totally even partitioning to explore the limits of scale-up. In Chapter 5, we will explore an alternative approach that uses the flexibility afforded by data replication to offer almost as good performance while restor-

[1]It is worth noting that the use of virtual nodes can also increase the effects of churn by giving each physical node more neighboring nodes. However, it does not increase the number of data items that must be moved, only the number of (now much smaller) sections of the key space that must be moved.

ing partitioning resiliency.

Additionally, our partitioning approach assumes that all nodes are equally powerful, as they are assigned the same fraction of the key space and therefore approximately the same amount of data. This assumption holds true for the experiments (with one explicit exception) in this chapter. The virtual node approach can compensate for this by assigning more virtual nodes to more powerful or better connected nodes, making them more likely to own a larger fraction of the key space. The techniques of Chapter 5 will also consider node heterogeneity. For the rest of this chapter, however, we focus on the simpler case of totally even partitioning over totally homogeneous nodes.

Data Retrieval

As mentioned above, a traditional DHT node maintains a routing table with only a limited number of entries (typically logarithmic in the number of nodes). This reduces the amount of state required, enabling greater scale-up, but requires multiple hops to route data. Recent peer-to-peer research in Gupta et al. (2004) has shown that storing a complete routing table (describing the partitioning of the key space among all nodes) at each node provides superior performance for up to thousands of nodes, since it provides single-hop communication in exchange for a small amount of state; we therefore adopt this approach. Our system requires a reliable, message-based networking layer connection with flow control. We found experimentally that, for scaling at least to one hundred nodes, maintaining a direct TCP connection to each node was feasible. With the use of modern non-blocking I/O, a single thread easily supports hundreds or thousands of open connections. For larger networks, a UDP-based approach could be developed to avoid the overhead of maintaining TCP's in-order delivery guarantees, as all of the techniques described here are independent of message ordering.

Node Arrival and Departure

Traditional DHTs deal with node arrival and departure through background replication. Each data item is replicated at some number of nodes (known as the *replication factor*). In Pastry, for example, for a replication factor r, each item is replicated at $\lfloor \frac{r}{2} \rfloor$ nodes clockwise from the node that owns it, and the same number counterclockwise from it, leading to r total copies. In the ring of Figure 4.2a, if $r = 3$, each data item that is owned by node n_1 will be replicated to n_4 and n_2 as well. When a node joins, background replication slowly brings all data items that a node owns to it, as they must

be stored at one of its neighbors. If a node leaves, each of its neighbors already has a copy of the data that it owned, so they are ready to respond to queries for data stored at the departed node. A similar approach works in the totally even partitioning scheme we adopt.

This approach makes an implicit assumption that all of the state at the nodes is stored in the DHT, and therefore that any node that has a copy of a particular data item can handle requests for it. If a node joins or fails, certain requests will suddenly be re-routed to different nodes, which are assumed to provide identical behavior (and hence do not get notified of this change). This does not work in the case of a distributed query processor, where in addition to persistent stored data there may be distributed "soft state" that is local to a query and is not replicated; this includes operator state, such as the intermediate results stored in a join operator or an aggregator. If data for a particular range is suddenly rerouted from one node to another, tuples might never "meet up" with other tuples they should join with, or data for a single aggregate group may be split across multiple nodes, causing incorrect results.

To solve this problem, our system works on *snapshots* of the routing table, which for us is the complete partitioning of the key space, since all nodes have global knowledge of other nodes' owned partitions. When a participant initiates a distributed computation, it sends out a snapshot of its current routing table, which all nodes will use in processing this request. Therefore, if a new node joins in mid-execution, it does not participate in the current computation (otherwise it may be missing important state from messages prior to its arrival). If a node fails, the query processor can detect what data was owned by the failed node, and thus can reprocess this state (this is discussed in Section 4.4).

Our system must still handle replication of base data, which is done in a manner very similar to that of Pastry; each data point is replicated at $\lfloor \frac{r}{2} \rfloor$ nodes clockwise and counterclockwise from the node that owns it. This ensures that data can survive multiple node failures, and that in the event of a node failure, the nodes that take over for a failed node have copies of the base data for the sections of the ring they are newly responsible for. Unlike in Pastry, a single node arrival or departure will cause all the ranges in the range to change slightly; this causes a membership change to be more expensive, but we are assuming reasonable bandwidth and less frequent failures. With smaller numbers of fairly reliable nodes, the performance benefits of uniform distribution likely outweigh the costs of extra shipping.

Currently we only replicate data as it is inserted into the DHT. This has been sufficient for the development and experimental analysis of our system, since we inserted data before any node fail-

ures, and failed few enough nodes that data was never lost. For completeness we plan to implement the Bloom filter-based background replication approach of the Pastry-based PAST storage system described in Druschel and Rowstron (2001), which can be directly applied to our context.

Using the Substrate on Cloud Services

One of the goals of our work is to be able to scale not only across the participants in the Orchestra system, but also, especially as query load increases, to be able to flexibly expand to incorporate cloud computing nodes. As we describe later, Orchestra can employ Amazon's Elastic Computing Cloud (EC2), which provides virtual Linux nodes upon which our query processor can be easily deployed. EC2 has several regionally distributed host sites with excellent connectivity, and we can quite efficiently migrate and replicate data to the EC2 nodes and bring them up to speed.

4.3 Versioned Data Storage

Recall from our earlier discussion that Orchestra supports a batched publish/import cycle, where each participant stores its own updates in the CDSS, disjoint from all others. There is no need for traditional concurrency control mechanisms, as conflicts among concurrent updates are resolved during the import stage (via reconciliation) by the participant. We therefore do not focus on such techniques. While this work is motivated by the needs of Orchestra, the system described here supports general reliable storage of relational data, provided concurrency control is not needed.

However, there is indeed a notion of global consistency. We assign a logical timestamp (*epoch*) that advances after each batch of updates is published by a peer. When a participant performs an import or poses a distributed query, it is with respect to the data available at the specific epoch in which the import starts. The participant should receive the effects of *all* state published up to that epoch, and no state published thereafter (until its next import). The current epoch does require some additional synchronization. As mentioned in Section 3.4, a distributed counter, maintained through standard techniques from distributed systems such as Paxos (Lamport, 1998) or PBFT (Castro and Liskov, 2002), is necessary for complete consistency. However, such "heavyweight" distributed consensus protocols are only needed once during each publish/import cycle. One an epoch is associated with this operation, all writes and reads can be relative to this epoch, using the techniques we describe here.

Of course, in order to support queries over versioned data, we must develop a storage and access layer capable of managing such data. There are several key challenges here:

- Between database versions, we want to efficiently reuse storage for data values that have not changed.
- We must track which tuples belong to the desired version of a database. Such metadata should be co-located with the data in a way that minimizes the need for communication during query operation.
- Each tuple must be uniquely identifiable using a *tuple identifier* that includes its version. Yet, for efficiency of computation, we must partition data along a set of key attributes (as with a clustered index). It must be possible to convert from the tuple ID to the tuple key, so that a tuple can be retrieved by its ID; therefore a tuple's hash key must be derived from (possibly a subset of) the attributes in its ID.

We maintain all versions of the database in a log-like structure across the participants: instead of replacing a tuple, we simply update our records to include the new version rather than the old version, which remains in storage. Disk space is rarely a constraint today, and the benefits of full versioning, such as support for historical queries, typically outweigh the drawbacks. We distribute this log partitioned along (possibly a subset of) a tuple's key attributes. Tracking temporal information is an integral part of *temporal databases*, where it was surveyed in Özsoyoglu and Snodgrass (1995). Associating versioned (or timestamped) information with a key is a standard approach in both peer-to-peer systems and databases. Examples include Haeberlen et al. (2005) in the former and Stonebraker (1987) and Buneman et al. (2002) in the latter.

Each node, therefore, may contain many versions of each tuple. If the set of nodes is in flux, nodes may come and go between when a tuple is inserted or updated and when it is used in a query; therefore, a node may not have the correct version of a particular tuple. We assume that background replication is sufficient to ensure that each tuple exists somewhere in the system, but that it may not exist where the standard content-addressable networking scheme can find it. We therefore need to be able to determine, for a particular epoch, the collection of tuple IDs that are present in a relation. Once the system has this information, scanning a relation becomes relatively simple. The tuple IDs are partitioned using the DHT routing table, and sent to the nodes that own them. Those nodes then

probe against persistent storage to find the full tuples associated with each ID. If the node has data for the tuple ID, it scans that tuple. Otherwise, it must have stale data for that key, or be entirely missing it, due to replication lag and network churn. The node that should own the tuple therefore searches outwards from the current node, looking first at nodes nearby in the key space, and then at those farther away, until it finds a copy of the data for that ID. If the tuple is eventually found in the system, it is copied to the node that should own it and scanned there to preserve data partitioning; otherwise we have confirmation that data has been lost due to insufficient replication. Such an approach will never suffer from silent failure.

A key property we adopt from CFS (Dabek et al., 2001) is that, once there is enough information to begin a request, it is always clear what data *should* be present in the distributed storage layer. In our case, this means that the current epoch has been determined by one of the distributed protocols discussed above; in CFS, it means that the root block has been retrieved, which determines which versions of all files will be used. In either case, stale data will never be retrieved. If expected data is not found at the node that should own it, this is likely due to network churn. The request can either be retried after background replication has moved state around, or the system can proactively try to retrieve the missing state from other nearby nodes. Our query-oriented approach, described above, attempts to proactively retrieve the necessary data.

The key feature of our approach to reliable storage are the two complementary storage layers. *Primary storage* contains a log of all versioned tuples, tagged with version numbers. *Secondary storage* contains what we term the *index*, which maps from versions of a relation to versions of particular tuples that are present in that relation. Implementing primary storage directly over raw distributed hash table is an obvious choice. As we require that the hash attributes be a subset of the primary key, it is possible to determine which node should store the full version of a tuple given its ID. In this thesis, we do not consider alternate ways of storing the versioned tuple log. There are, however, many possible ways of storing the index. At the limit, one could store a list of tuple IDs as a single object in the DHT. This could be very expensive to update, however, since any update to a relation would require the (slightly) modified list of tuple IDs to be rewritten in its entirety.

We have experimented with several *hierarchical* implementations of the index, which introduce a level of indirection to allow unmodified portions of the index to be used by a relation at multiple epochs. We were initially inspired by filesystem i-nodes, the CFS filesystem (Dabek et al., 2001), and log-structured filesystems, where for append operations and small changes, the page-level data in a

large file mostly remains unchanged. Such schemes all make use of a versioned system for tracking the contents of a file, which greatly resembles our index. With this prior work in mind, we decided to divide a relation into multiple *index pages*, each of which is given a unique ID. A *master record* for a relation records which index pages are used by that relation for a particular epoch. When a relation is updated, only those index pages that are changed need to be rewritten, and a new master record written out that refers to a mixture of old and new pages. The new versions of updated tuples need also be written out to primary storage, of course.

This leaves the question of how to organize tuple IDs onto index pages. If they are organized in a particular way, we may be able to exploit that when updating or querying a relation. One option, of course, is not to organize the index pages in any particular way. This would make deleting or updating a relation expensive, since all index pages might have to be searched to find the page that would need to be updated when updating a tuple. It would, however, allow the system to cluster frequently updated keys on the same index page, maximizing the reuse of index pages by ensuring that as many index pages as possible remain unchanged from epoch to epoch. A second option is to sort the tuple IDs by their primary key in the relation (i.e. by their logical keys), and partition them into pages based on that ordering, so each page holds a continuous range for the primary key. Then it is easy to find which page contains a key when updating or deleting a tuple, eliminating the potentially lengthy search described above. This approach will also potentially allow range predicates over the primary key to be executed at the level of the index, eliminating tuples as early as possible in a scan. A third option is to sort the tuple IDs by their ID in the DHT key space, and partition them into pages so that each page holds a contiguous range in the key space. This has the same benefits for updating as clustering the tuple IDs by their primary key, but does not allow range predicates over the primary keys. It does, however, allow the pages to be stored near to the tuples they reference in the DHT key space. If the number of pages is large enough and the number of nodes is small enough, then with high probability a node will own the index pages for the tuples it owns. Preliminary experiments showed that the benefits of the last approach for scan performance are very high, and that performance was much better than for the second option. We therefore cluster index pages by DHT ID in our implementation, and all experimental results shown in this chapter use this approach.

Figure 4.3 shows the main data structures used to ensure consistency. All data structures are replicated using the underlying network substrate, so failure of any node will cause all of its functionality to be assumed transparently by one or more neighboring nodes. All nodes in the system perform a

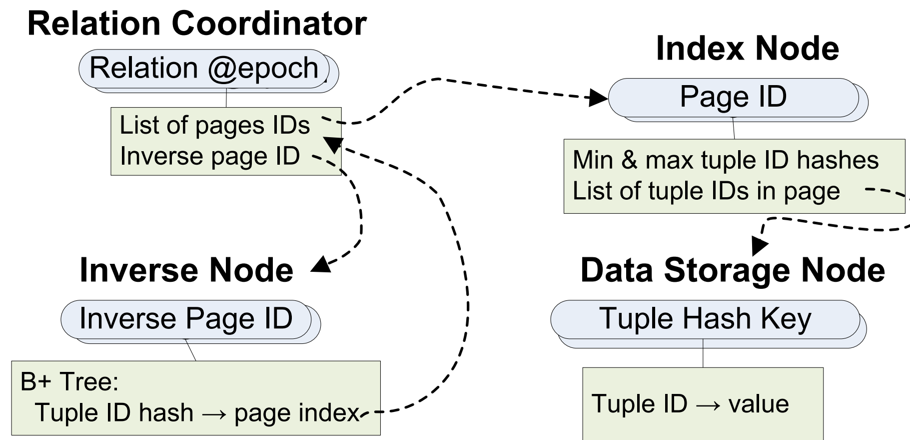


Figure 4.3: Storage scheme to ensure version consistency and efficient retrieval. Rounded rectangles indicate the key used to contact each node (whose state is indicated with squared rectangles).

number of tasks based on the data stored at them. In its capacity as a *data storage node*, each participant holds a portion of the versioned tuple log by storing a mapping from tuple IDs (recall that this is the primary key plus a version identifier) to full tuples; this is the primary storage mentioned above. The hierarchical index is shared among three types of nodes. The *relation coordinator* for a particular relation at a particular epoch holds the master record for that relation and epoch, a list of the page IDs used in that version of the index. It also holds the *inverse page ID*, which identifies a B+ tree that allows the system to quickly determine which page a particular primary key would fall onto. This is split apart from the master record because small changes to a relation over time may not necessitate changing the boundaries in the DHT key space between pages, and therefore the lookup tree can be reused; it references indices in the list of page IDs instead of actual page IDs. An *index node* holds index pages, keyed by page ID. Each page holds the tuple IDs for a contiguous range of the DHT key space. These tuple IDs can used to probe the data storage nodes for the full version of each tuple in the relation. Recall that we place the index node entry at the same node as the tuples it references, by storing the index page at the middle of the range of tuple keys it encompasses. This is why the network substrate, as discussed in Section 4.2, assigns a large, contiguous region in the key space to each node; it means that the vast majority of tuple keys are never sent over the network. If each node is responsible for many smaller ranges, this is no longer the case, and performance suffers. Thus for most tuples the same participant is both the index node and the data storage node.

Our scheme is designed to improve update performance and reduce storage overhead by effi-

ciently supporting relatively small changes to tables. Modifying a tuple in a relation requires us to look up the page holding the old version of the tuple using an inverse node, modify that page to include the ID of the new tuple, and write out that modified page as the new index page for the region of the table surrounding the updated tuple. The entire contents of the new tuple must also be written out to the network. The system then creates a new version record linking to the updated index page, and all of the unaffected pages from the previous version. No modifications happen at the data storage nodes except to add the new version of the modified tuple; the previously used versions of all other tuples continue to be referenced by the reused or updated index pages.

Example 1. Suppose we have three participants, each storing a partition of a simple, one-table database, $R(\underline{x}, y)$, where x is the key and y is a non-key attribute. Node n_1 is responsible for the range [0x00 ... ,0x55 ...], n_2 for [0x55 ... ,0xAA ...], and n_3 [0xAA ... ,0x00 ...]. In this example, the tuple ID is the key attribute of a tuple and the epoch in which it was last modified, e.g., $\langle f, 1\rangle$ for $R(f, a)$. The index page ID consists of the relation name, the epoch in which it was last modified, and a unique identifier for that relation and epoch, such as $\langle R, 1, 1\rangle$ for the second index page created for relation R during epoch 1. It also includes the hash ID where the index page is stored.

In the first epoch (epoch 0), a participant inserts the tuples $R(a, b)$ and $R(f, z)$; system state after this operation is shown in Figure 4.4a. In epoch 1, someone inserts $R(b, c)$, $R(e, e)$, and $R(c, f)$ while also changing $R(f, z)$ to $R(f, a)$; this is shown in Figure 4.4b. In epoch 2, someone inserts $R(d, d)$. The final state of the system is shown in Figure 4.4c. All of the structures stored in the system at epoch 2, included state from previous epochs, is shown in Figure 4.5.

Pseudocode for performing a lookup appears as Algorithm 4.1. Retrieval starts at the relation coordinator for the requested epoch, from which a list of index nodes can be obtained. It sends a scan request to each index node, along with any predicates that can be evaluated over the key attributes present in the tuple IDs (the so-called sargable predicates). The index nodes apply all such predicates to the list of tuples for each index page, and requests that the matching tuples be retrieved. This operation is highly parallelizable; the only operation done at a single node is the sending of the scan requests, which is very fast.

Example 2. Figure 4.6 shows how the lookup procedure works for our example instance. First, the lookup request from n_2 for relation R at epoch 2 is hashed to find the node (in this case, n_1) that is the relation coordinator for the relation at the desired epoch. The data stored there contains the list

Data Node	Tuple	Tuple ID	Hash ID	Index Page
n_1	R(f,z)	⟨f,0⟩	0x00…	
	R(a,b)	⟨a,0⟩	0x20…	
n_2				⟨R,0,0⟩@0x80…
n_3				

(a) State at epoch 0

Data Node	Tuple	Tuple ID	Hash ID	Index Page
n_1	R(f,a)	⟨f,1⟩	0x00…	
	R(a,b)	⟨a,0⟩	0x20…	⟨R,1,0⟩@0x40…
n_2	R(b,c)	⟨b,1⟩	0x60…	
n_3	R(e,e)	⟨e,1⟩	0xD0…	⟨R,1,1⟩@0xC0…
	R(c,f)	⟨c,1⟩	0xF0…	

(b) State at epoch 1

Data Node	Tuple	Tuple ID	Hash ID	Index Page
n_1	R(f,a)	⟨f,1⟩	0x00…	
	R(a,b)	⟨a,0⟩	0x20…	⟨R,1,0⟩@0x40…
n_2	R(b,c)	⟨b,1⟩	0x60…	
	R(d,d)	⟨d,2⟩	0x80…	
n_3	R(e,e)	⟨e,1⟩	0xD0…	⟨R,2,0⟩@0xC0…
	R(c,f)	⟨c,1⟩	0xF0…	

(c) State at epoch 2

Figure 4.4: State of the versioned relation R at each of the epochs for Example 1. Data is partitioned across nodes by the key (the first attribute), which is a subset of the Tuple ID. Redundant copies of replicated data are not shown, nor are versions of tuples that only appear at previous epochs. The left brackets indicate which nodes a tuple is stored on, while the right brackets indicate which index page a tuple's ID is on.

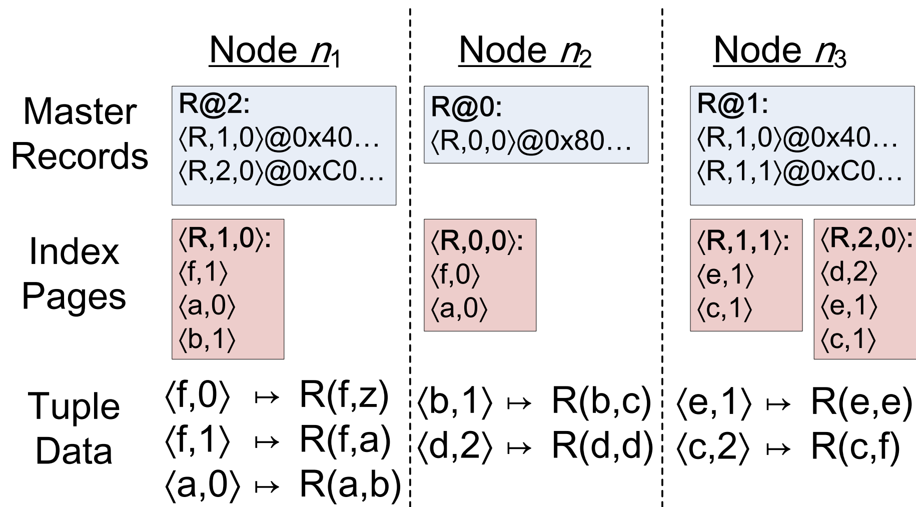


Figure 4.5: All of the state stored in the system for versioned relation R after all of the operations described in Example 1 have taken place.

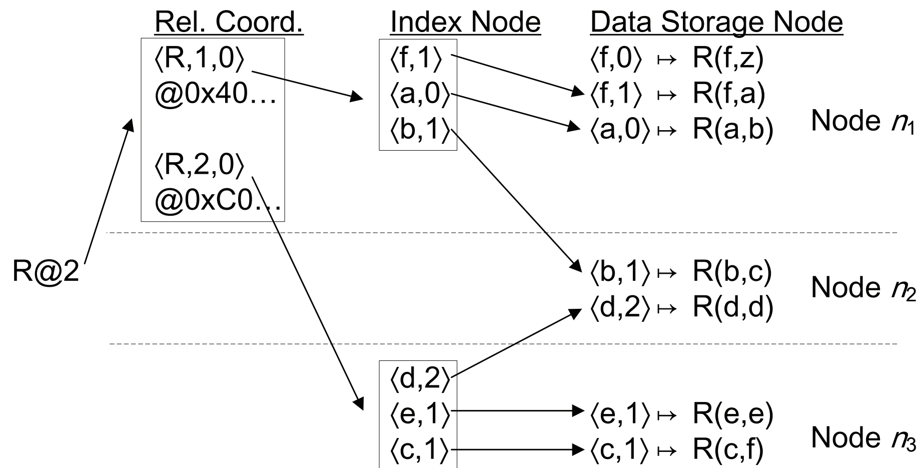


Figure 4.6: Lookup of relation *R* at epoch 2 for the example instance, as described in Example 2.

of index pages that contain the tuple IDs for that version of the relation. The request to scan those pages is sent to the index nodes that contain the contents of the pages, in this case n_1 and n_3. Those index nodes then send requests on to the data storage nodes that contain the full tuples (stored as a mapping from Tuple ID to full tuple) to scan the desired tuples given their IDs. The data storage nodes then retrieve the desired tuples and return them to the requester (not shown). Note that only two of the six Tuple IDs were actually sent over the network, due to the colocation of index pages and tuple data.

As mentioned before, this approach avoids any possibility of seeing stale data due to replication lag. Suppose that, for some reason, n_1 had not yet received a copy of the record for R at epoch 2. It would search other nodes nearby in the system until it found a copy before proceeding. Similarly, if n_1 had not yet received the data $\langle f, 1\rangle$, it would never simply return the data for $\langle f, 0\rangle$; it knows that data is stale because it does not appear in the index page. It would instead try to retrieve the full tuple for $\langle f, 1\rangle$ from the network before proceeding.

Algorithm 4.1 DistributedRetrieve$(R, e, f(\bar{k}))$
Input: R (relation), e (epoch), $f(\bar{k})$ (filter function over key $\bar{k}$)
Output: Matching tuples $t \in R$ satisfying $f(\bar{k})$.

1: *relCoord* $\leftarrow h(\langle R, e\rangle)$
2: Contact Relation Coordinator at *relCoord*, retrieve *pageIDList*
3: **for** *page* $\in$ *pageIDList* **do**
4: Ask Index node at $h(\langle e, (\mathit{page.max} + \mathit{page.min})/2\rangle)$ to scan page *page*
5: Index node retrieves *page* contents *Tuples*
6: Index node filters *Tuples* with $f(\bar{k}) \rightarrow$ *fTuples*
7: **for** $t \in$ *fTuples* **do**
8: Index node requests that Data Storage node at $h(t.\mathtt{key})$ scan the tuple t
9: Data Storage node sends t to node that requested scan, bypassing the Index node and Relation Coordinator
10: **end for**
11: **end for**

4.4 Reliable Query Execution

The prior section described how to achieve reliable access to mutable data in a distributed hash table. In this section, we explore how to perform reliable processing over this data. In combination, this will allow us to achieve correct and complete answers to queries, even under network and in the presence

of node failures. This is contrast to prior peer-to-peer query engines, such as PIER (Huebsch et al., 2005) and Seaweed (Narayanan et al., 2008), which were best-effort. Their focus was on scaling to large numbers of nodes, and do to so it was necessary to sacrifice strict consistency. In our work on query processing for collaborative data sharing, we can exploit the fact that we are limited to a smaller scale to ensure reliable processing. In the case of ORCHESTRA, this means reliable execution of the queries that implement the update store and perform update exchange. For other distributed applications, this means any "single-block" SQL query, i.e. any query consisting of joins, scalar function evaluation, predicate evaluation, and possibly a single level of aggregation.

As in this prior work on peer-to-peer query engines, we suffer from the general problems in wide-area distributed systems of higher lag times, constrained bandwidth, and generally bursty communication. Like those systems, we adopt a push-driven style of distributed query processing. The operators at each node either receive data directly from a local scan of persistent storage, or receive tuples as they arrive from other nodes in system. This ensures that as much processing is done as possible given the available data, and to enables flexible operator scheduling in the event of delays. Also like PIER, most operators are partitioned-parallel, using hash ranges as the partitioning function; a small fraction of each operator executes on each node. This gives the system high scalability, assuming a good hash function. Partitioned parallelism happens naturally for scans, given the data layout described in the previous section. For other operators, it necessitates repartitioning to ensure that tuples than need to end up at the same node do so. For example, in the case of a distributed join, the tuples must be rehashed on (possibly a subset of) the join attributes, to ensure that any tuples that should join are able to; similarly, for an aggregation, it must be a subset of the grouping attributes. All data is ultimately collected at the *query initiator* node, which may do final processing, such as the last stage of aggregation, or a final sort.

Hashing-Based Distributed Query Execution

A query plan consists of a number of operators. Most are implementations of standard relational operators, though a few are specialized to our storage layer or hashing-based partitioned-parallel databases. Our system implements the following operators:

Covering index scan retrieves data directly from the index nodes, if only key attributes are required, bypassing the data storage nodes.

Distributed scan executes at both index nodes and data storage nodes, similar to in Algorithm 4.1. The index nodes filter index pages to eliminate tuples that don't pass a predicate over the relation's key attributes, and then send the passing tuple IDs to the data storage nodes. The tuples from each index page are stored nearby on disk, and are retrieved using the tuple IDs in a single pass through the hash ID range for that page. Instead of being sent back to the query initiator, the resulting tuples are pushed through the query plan.

Select implements selection on intermediate results.

Project is the standard projection operator.

Join is a *pipelined hash join* (Raschid and Su, 1986).

Aggregate is a a blocking, hash-based grouping operator, which supports re-aggregation of partially aggregated intermediate results.

Ship sends the tuples it receives to the query initiator.

Rehash partitions its input among the system nodes by hashing on some subset of the tuples' attributes.

Compute-function performs scalar function evaluation, such as arithmetic or string concatenation.

Spool buffers its input tuples as the query initiator, so they can be retrieved once the query is finished.

The query is "driven" by some combination of the leaf-level scan operators described in the table — each is novel to our system, as it exploits the specific versioned indexing scheme used in our storage system. Such operators typically are run concurrently across all of the nodes in the system — each operating on a data partition stored at those nodes.

From there, the retrieved data may be passed locally through a series of pipelined operators, such as joins or function evaluation. Recall that most operators execute in a partitioned-parallel fashion, meaning that a logical operator in the query plan is represented by a collection of physical operators, one at each node, each executing over a portion of the data in the system. Processing continues using a given partitioning of the data (i.e. using a particular subset of tuple attributes as input to the hash function) until the data needs to be repartitioned to enable a different computation. This happens

using either a *ship* operator or a *rehash* operator. The ship operator sends the data it receives to the query initiator; this is needed to get the results to the spool operator that collects the results of a query. The rehash operator repartitions its input tuples by their hash IDs in the networking substrate and sends them to other nodes in the system. Rehashing is commonly used to enable joins or aggregation, when a relation needs to be re-partitioned on a join or grouping key. The rehash operator routes tuples to a destination node by first hashing the key using the SHA-1 hash function, then consulting the snapshot of the *query routing table* described previously.

Each operator sends an *end-of-stream notification* to its parent operator when it finishes executing. Scans can easily detect when they are done, and most other operators simply propagate an end-of-stream notification downstream after they receive it (perhaps first performing some final computation to produce results, as in a aggregate operator). However, detecting end-of-stream with the rehash operator is slightly tricky: it cannot complete until it has acknowledgment from all downstream nodes that they have received all of the data it sent. Once the spool operator (the root of all query plans that holds the query results) has received an end-of-stream notification, all other operators must also have finished, and so the query is complete.

We now show two examples of query plans for our system. Each is annotated with the intermediate state that might be generated during an execution of that plan. Example 3 shows how a distributed join takes place, and Example 4 shows how a distributed aggregation takes place.

Example 3. Continuing our example of versioned storage given in Example 1, let us consider the following query, which performs a self-join of the relation `R(x,y)`:

```
SELECT x, z
FROM R r1, R r2
WHERE r1.y = r2.x
```

An Orchestra query execution plan, overlaid with data from the example instance as it would flow through the plan, is shown in Figure 4.7; we assume that node n_1 posed the query and is therefore acting as the query originator. Each node begins by scanning two copies of `R`, one for `r1` and one for `r2`. The copy for `r1` is then repartitioned on `r1.y`. It then joins with `r2`, which is already partitioned on with `r2.x`, making this a valid distributed join. The resulting tuples are then sent to the query originator, where they are spooled. As mentioned previously, end-of-stream notifications will

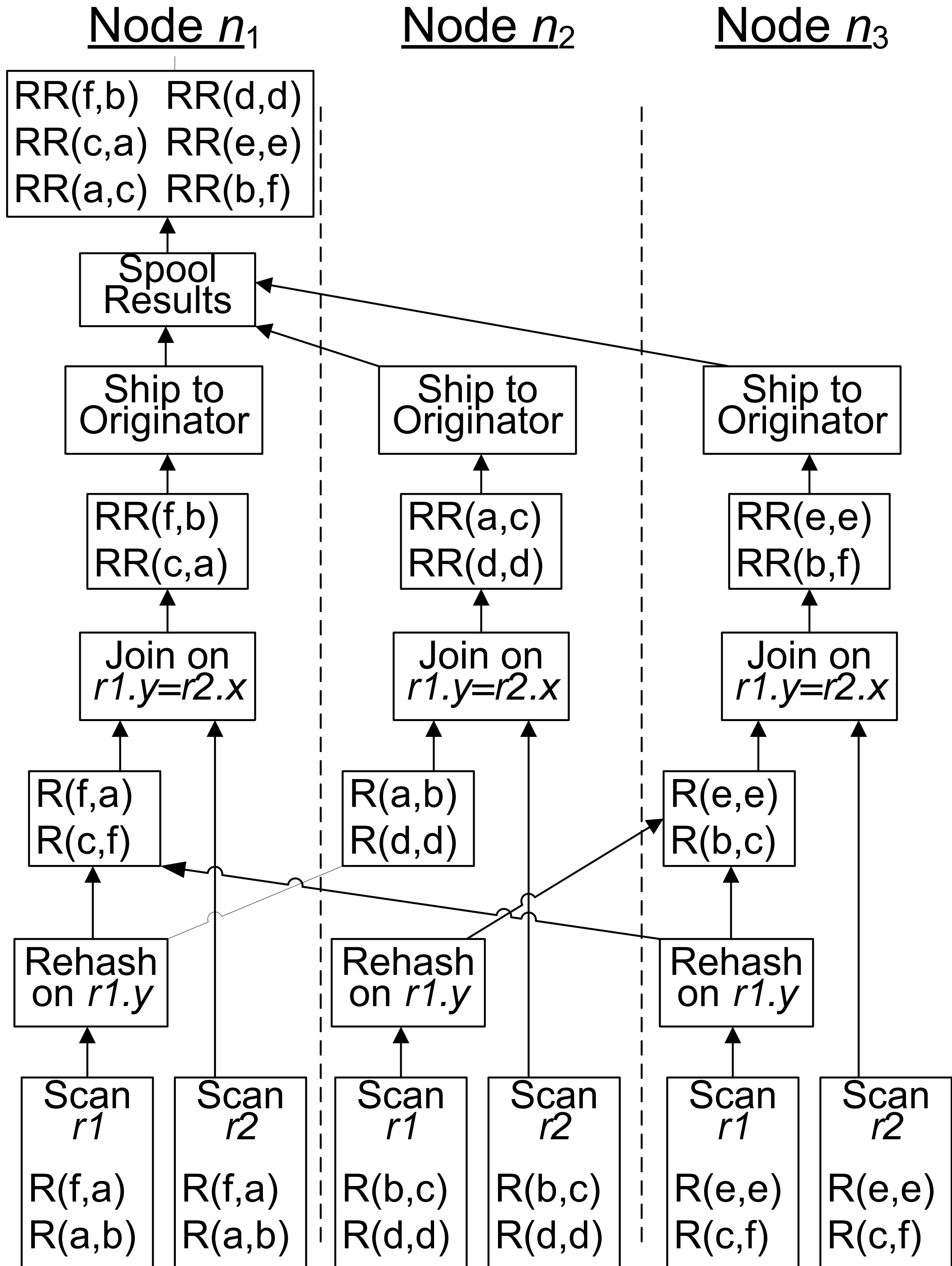


Figure 4.7: Distributed query plan for Example 3.

propagate up through the query plan from the leaves; when they reach the spool operator, the query results are complete.

Example 4. Now let us consider an aggregation query. Since the relation for the previous example contained no numeric data, let us instead consider the relation `StatePop(state,region, population)`. The primary key of `StatePop` is `state`, which is also the hash key used for partitioning. Suppose that node n_1 wished to execute the following query:

```
SELECT region, SUM(population)
FROM StatePop
GROUP BY region
```

As before, we present a distributed execution plan, overlaid with data from a possible instance; this is shown in Figure 4.8. This plan performs a distributed scan of the `StatePop` relation, which produces data partitioned by state ID. The data is then repartitioned by hashing on the `region` attribute. Note that since there are only two regions present in the data, at least one node will not receive any data; in that case, only n_1 and n_3 receive data. One the aggregate operators receive end-of-stream notifications from all of the scan operators, they know that they will receive no more input and can produce their output. These tuples are then shipped to n_1, the query originator, which collects the results to the query.

Of course, the queries presented in Examples 3 and 4 are relatively simple. The each contain only one repartitioning of the data through rehashing. More involved queries, containing multiple joins, possibly combined with aggregation, will make considerably more use of the rehash operation to distribute intermediate state so it can be used in a series of operators. In the examples, there were only a few possible ways of translating the query into a query plan, and the ones chosen are intuitively among the best. With more complex queries, where operations may be reordered and there may be multiple partitioning schemes that make sense, the query optimizer plays a critical role in choosing a good plan, based on data statistics and data about the participants' capabilities. We will discuss in more detail how it does this in Section 4.4.

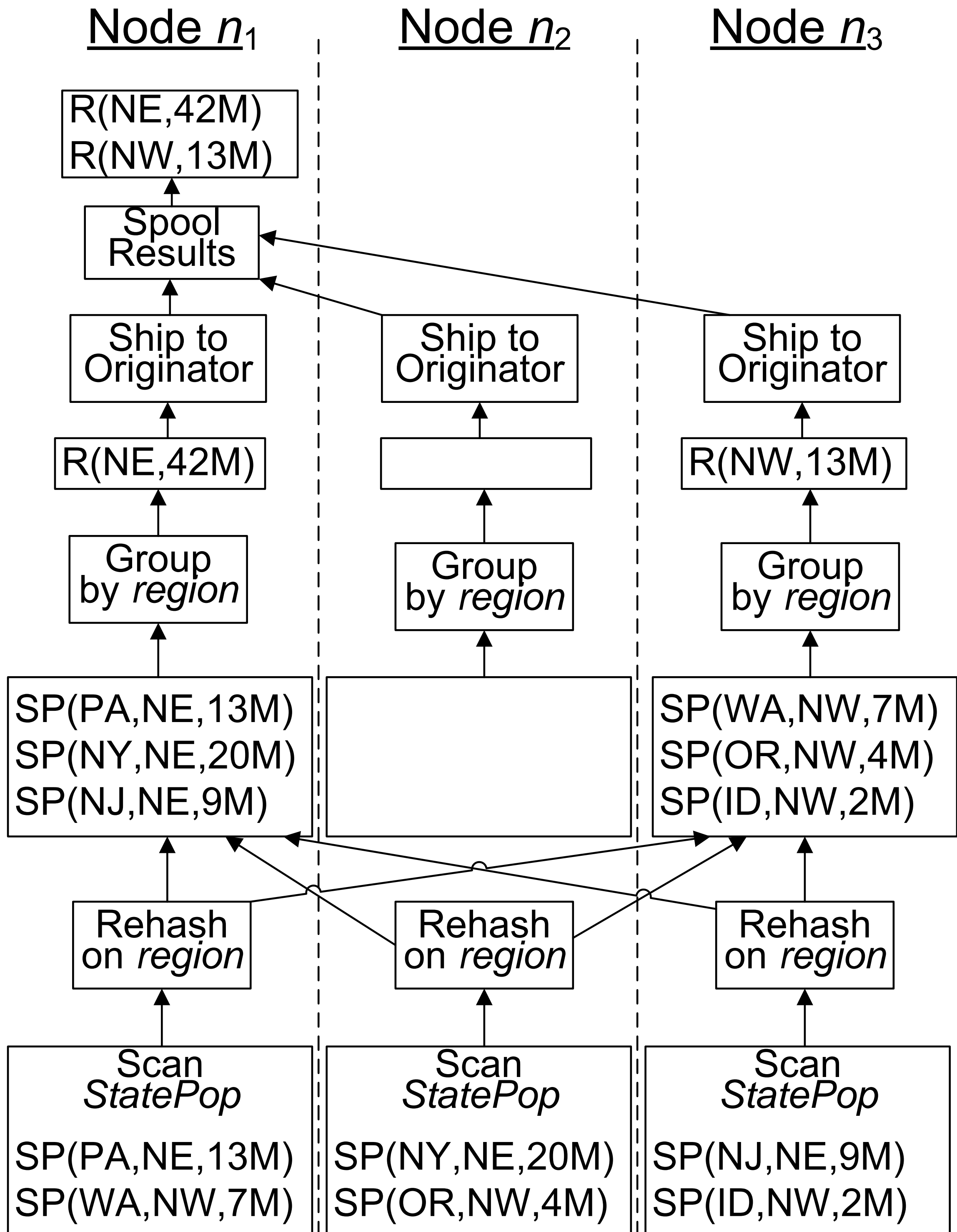


Figure 4.8: Distributed query plan for Example 4.

Architecture for Performance and Failure Detection

Several aspects of our query processing architecture are enabled by our custom hash-based substrate. Prior work has used existing "off the shelf" DHTs. For example, PIER uses Bamboo (Rhea et al., 2004), while Seaweed uses Pastry. However, we additionally develop several techniques at the query execution level that are vital for performance and correctness.

First, for **efficiency**, the query processor benefits from the fact our substrate uses TCP to manage connections between machines. This allows for automatic flow control in the event of a congested network. Of course, we could have alternatively used UDP, and implemented flow control via periodic handshaking. However, with the use of non-blocking I/O routines (to avoid the large numbers of threads inherent in using blocking I/O over many channels), we saw no evidence that maintaining even hundreds of open TCP connections imposed any significant overhead.

Second, for rapid **failure detection**, we use frequent UDP-based pings. While existing DHTs also use a so-called "heartbeat" to detect node failure, they typically do not detect failures quickly enough for our purposes. We need these notifications of node failure to percolate quickly up to the query execution layer, so it can compensate for them in some way. In existing DHTs, failover is often totally transparent to the application execution above the networking layer.

Third, for **failure recovery**, the query processor is given direct information about the state of the routing tables. A *snapshot* of the routing tables is taken by the query initiator as it invokes the query; recall that, as in ORCHESTRA all nodes have global knowledge, this is a complete snapshot of the partitioning of the DHT key space. This snapshot is disseminated along with the query plan to all nodes, in order to ensure absolute consistency of the routing tables. If one or more nodes fail in the middle of execution, the difference in the routing tables is reported back to the query initiator, such that it can incrementally recompute *only* the lost portion of the query state. We describe this feature in detail in Section 4.4.

Fourth, for **performance**, the query processor *batches* tuples into blocks by destination, compressing them (using lightweight Zip-based compression) and marshalling them in a format that exploits their commonalities. This makes query processing much more efficient than if it were built over a DHT with many smaller messages, and reduces CPU and bandwidth use.

Finally, for **correctness**, each tuple is annotated with information about which source nodes supplied the data from which the tuple was derived. This is used to prevent duplicate answers when

recovering from a failed node.

Handling Node Membership Changes

The major challenge of reliable query processing is how to handle changes to the node set. Recall from Section 4.2 that the query initiator takes a *snapshot* of the routing table (which, for us, is the complete key space partitioning) in the system during query initiation. It disseminates this snapshot along with the query plan so all machines will use a consistent assignment between hash values and nodes. The query initiator is itself deciding which participants are to participate in the query execution; this is similar to the work performed by a distributed group membership protocol, such as in Reiter (1996) or Birman and Joseph (1987), for example.

Node arrival

Suppose a node joins the system in the midst of execution. In a DHT, such a change immediately affects the routing of the system — and begins forwarding messages to the new node, which may not have participated in any prior computation. In principle, one might develop special protocols by which the new node would be "brought up to speed" by its neighbors. However, this becomes quite complex when multiple nodes join at different times. Instead, we let the query complete on its initial set of nodes, and only make use of the new node when a fresh query (with a new routing table snapshot) is invoked. This approach provides simplicity and avoids expensive synchronization.

Node departure or failure

Our use of TCP connections between nodes is often adequate to detect network partition or a node failure; we assume complete system failure, or at least a crash of our software running on the system, rather than incorrect operation. If a sending node (and query operator) drops its connection before sending an end-of-stream message, or a receiving node drops its connection before query completion, then this represents a failure. As mentioned previously, the system also performs periodic ping operations in the background to detect a machine to which the TCP connection (at least appears to) remain open, but which on which our distributed query processor is no longer successfully running, perhaps due to a software crash. In either case, ignoring the failure at the node or connectivity failure

between the node and the rest of the system will lead to missing or possibly incorrect answers. This leads us to the problem of recomputation, described in the next subsection.

Recovery from Failure

Our system supports two forms of recovery from failure. One option, upon detecting a node failure, is to terminate and and *restart* any in-process queries. Assuming low failure rates, we will ultimately get the answers this way; the indexing structure described in Section 4.3 will ensure that the scans in the restarted query are correct, and therefore, unless another node fails, the query will complete with accurate results. This approach is straightforward to implement in Orchestra, since we can detect which queries are still in-flight — in contrast to systems like PIER.

When failure during query execution is more likely, as in longer-lived queries running on large numbers of nodes, better performance (i.e. correct query answers are returned sooner) might be obtained by performing *incremental* recomputation, where we only repeat the computations affected by the failed node, using a different node that has data replicated from the failed one. The key challenge here is that simply recomputing will likely result in the creation of some number of duplicate tuples — which in turn will either lead to duplicate answers or (in many cases) to incorrect aggregate results.

After a failure, any derived state in the system that originated from the failed nodes is likely to be inconsistent, due to propagation and computation delays. We can re-invoke the computation from the failed nodes and then remove duplicate answers, or instead we can remove all state derived from the failed nodes' data before performing the recomputation. We adopt the latter approach due to the difficulty of detecting which tuples are duplicates. As was hinted at previously, this means we must track which intermediate and final results are derived from data processed at one of the failed nodes. We tag each tuple in the system with the set of nodes that have processed it (or any tuple used to create it), and maintain these sets of nodes as the tuples propagate their way through the operator graph. As we validate experimentally, this can be done with minimal overhead. Granted, even some overhead will probably increase the average time to query completion, if failures are rare; however, in many instances we feel that having less variable query execution times, and in particular reducing the maximum time it may take to execute a query, is worth a slight increase in the average execution time.

We divide incremental recomputation into four stages.

1. **Determine change in assignment of ranges to nodes.** When a node or set of nodes fail, other nodes "inherit" a portion of the hash key space from failed nodes. The query initiator computes a new routing table from the original one, assigning the ranges owned by the failed nodes to remaining ones. If the failed nodes' data is available on more than one replica, the initiator will evenly divide among them the task of recomputing the missing answers.

2. **Drop all intermediate results dependent on data from the failed nodes**. To prevent duplicate answers, we scan the internal state of all operators and discard any tuples that are tagged as having passed through a failed node (we term these *tainted* tuples). It is critical that any state *not* dependent on the failed nodes remains available. This is easy to accomplish with join operator state. For aggregate operators, we partition each group into sub-groups that summarize the effects of all of the tuples for each possible set of contributing nodes, and drop the sub-groups for failed nodes. While the number of subgroups is exponential in the number of rehashes (for n nodes and m rehashes, $\sum_{k=1}^{m+1} \binom{n}{k}$), this number is typically small; critically, it does not depend on the number of input tuples. Tuples that are in flight between operators (or crossing the network) must also be filtered in this way.

3. **Restart leaf-level operations for the failed nodes' hash key space ranges.** We restart leaf-level operations such as tablescans, re-producing any data that would have originated at the failed nodes. As the data propagate through the system, they will be re-processed against the data from other nodes, generating all join and grouping results dependent on them.

4. **Re-create data that was sent to the failed nodes' hash key space ranges.** Additionally, any data that was sent *to* a failed node was either lost when the node failed or has become tainted by passing through the node and will therefore be discarded. Now all data that was to have been sent to the failed nodes must be retransmitted. If an operator maintains an in-memory snapshot of all data necessary to re-produce its answers (as with a pipelined hash join) this is relatively efficient. For more costly operations such as tablescans, we add a *cache* of their output data, at the downstream *rehash* or *ship* operator. It is easy to detect which of the reproduced tuples would have been sent to a failed node by consulting the query's original routing table.

Perhaps the most difficult task in recovery is avoiding race conditions that lead to subtly incorrect query results. We have chosen to divide computation into *phases* corresponding to the initial execution, followed by successive incremental recovery invocations. Each tuple gets tagged with a phase. As each stateful operator processes a recovery message, it purges tainted data and increments its phase counter. All tuples it (re)produces are in this new phase. This allows the system to differentiate between old, in-flight data from a failed node and new, recomputed results from recovery.

Query Optimizer

The focus of this chapter is on the distributed execution engine of ORCHESTRA, but we briefly describe its optimizer. It currently handles single-block SQL queries, including function evaluation and aggregation. It adopts the Volcano (Graefe, 1990) transformational model, using top-down enumeration of plans with memoization, and employing branch-and-bound pruning to discard alternative query plans when their cost exceeds the cost of a known query plan. Our optimizer considers bushy as well as linear query plans. It relies on information (previously computed and stored) about machine CPU and disk performance, as well as pairwise bandwidth. The optimizer estimates costs by assuming that each horizontally partitioned relation will be evenly distributed by the storage layer across all nodes. It then estimates the cost of a subplan by considering the cost at the slowest node or link that must be used at each stage — in a sense estimating the worst-case expected completion time of each operation.

4.5 Experimental Evaluation

We begin by briefly describing our implementation, which has been under development for more than two years. Our execution engine is implemented in approximately 50,000 lines of Java. It uses BerkeleyDB Java Edition 3.3.69 for persistent storage of data. We conducted most experiments on a 16-node cluster of dual-core 2.4GHz Xeon machines with 4GB RAM running Fedora 10, connected by Gigabit Ethernet. To study performance at scale, we used up to 100 2GHz dual core nodes from Amazon's EC2 cloud computing service.

Workload

Queries that are generated from schema mappings, as in data exchange and collaborative data sharing systems, are primarily select-project-join queries that vary from domain to domain, and are seldom publicly available. Alexe et al. (2008) developed a new benchmark suite, STBenchmark, which creates synthetic data exchange schema mappings along a variety of dimensions. We ran the STBenchmark instance and mapping generator with the default parameters, but with the nesting depth set to zero to produce relational data. We varied the size of each generated relation from 100K to 1.6M tuples (the maximum the ToXGene data generator could produce due to memory constraints). Except for one field, all STBenchmark tables are wide relations containing many 25-character variable length strings (which are not necessarily representative of typical data exchange settings). Nonetheless, we selected a representative subset of the STBenchmark mapping scenarios to study:

1. **Copy**, which retrieves an entire 7-attribute relation,
2. **Select**, which retrieves the tuples from a 6-attribute relation that satisfy a simple integer inequality predicate,
3. **Join**, which combines a 7-, a 5-, and a 9-attribute relation by joining them on two attributes,
4. **Concatenate**, which retrieves a 6-attribute relation, concatenates three of those attributes together, and returns the result along with the remaining three attributes, and
5. **Correspondence**, which retrieves a 7-attribute relation and uses a correspondence table to add an integer-valued ID based on two of the input attributes to the result.

The last query used a Skolem function (ID generator) in the output, which we replaced with a value correspondence table, which is typical of data integration settings.

To add diversity and scale to our data and queries, we also experimented with the standard TPC-H OLAP benchmark for the following reasons:

1. it scales to a variety of sizes, enabling us to consider dataset scalability,
2. it contains a diverse set of queries, enabling us to identify different performance factors, and
3. it is a well-understood and standard benchmark for comparison.

We used the standard TPC-H data generator to create source data at several scale factors, and we selected the TPC-H queries meeting the single-SQL-block requirement of our optimizer. We distributed the 8 TPC-H tables by partitioning on their key attribute (first key attribute, if more than one attribute was present). Two of the tables, *Nation* and *Region*, were small enough that we replicated them at each node; together they take up less than 3KB on disk. We use TPC-H queries 1, 3, 5, 6, and 10, and measure running time to completion of the full query. Queries 1 and 6 are aggregation queries over the *Lineitem* table; Q1 performs a distributed aggregation followed by re-aggregation at the query coordinator, while Q6 only performs an aggregation at the coordinator. Queries 3, 5, and 10 are 3-way, 6-way, and 4-way joins, respectively, followed by aggregation.

While we feel that the distributed storage and computation approaches described in this chapter are potentially applicable to a variety of problems, they were developed with a distributed implementation the Orchestra CDSS in mind. In such an implementation, the query and storage layer would provide a distributed implementation of the update store, the log of all transactions. It is also responsible for executing the SQL queries that arise from the update exchange process, and for evaluating trust conditions over translated transactions. Given that there are no actual collaborative data sharing instances in production use, and in fact the implementation of Orchestra over this new storage layer is not complete, we feel that the data integration queries from STBenchmark and the many-way joins and aggregations provided by TPC-H give a representative sample of the kinds of queries that our system will eventually have to execute.

All measurements were taken after results converged to a stable range of values; this is done to ensure warm caches and to avoid invoking the Java JIT compiler, which otherwise adds a large amount of noise. We present the mean of five runs, and show 95% confidence intervals for the mean for all data points

Performance in the Local Area

We first study the performance of our engine over our cluster's local network (running at the full Gigabit speed), to see how the architecture scales.

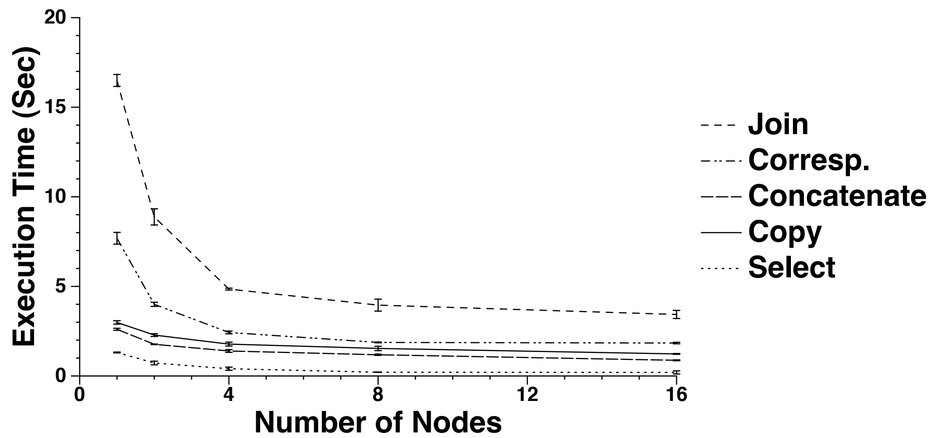


Figure 4.9: Running time: STBenchmark, 800K tuples/relation, 1-16 nodes. Performance continues to improve as the number of nodes increases.

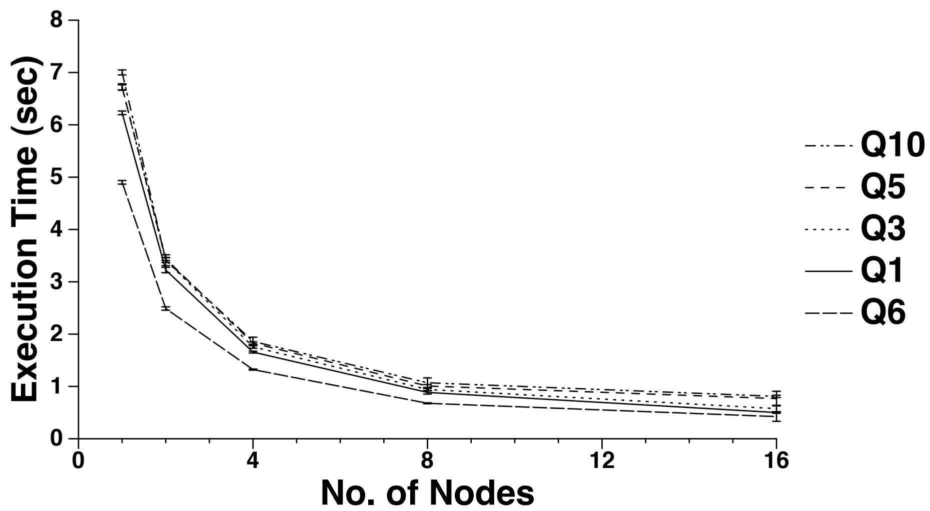


Figure 4.10: Running time: TPC-H Scale Factor 0.5, 1-16 nodes. Execution times continue to decrease as the number of nodes is increased.

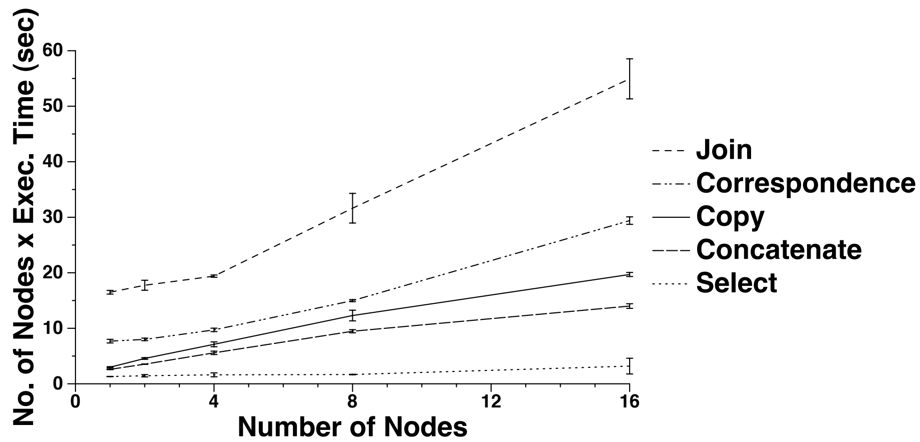


Figure 4.11: Normalized running time: STBenchmark, 800K tuples/relation, 1-16 nodes. The normalized time shown, the product of execution time and the number of nodes, would be flat if the system scaled perfectly. As shown, join queries scale worse than other queries, but all benefit from adding additional nodes.

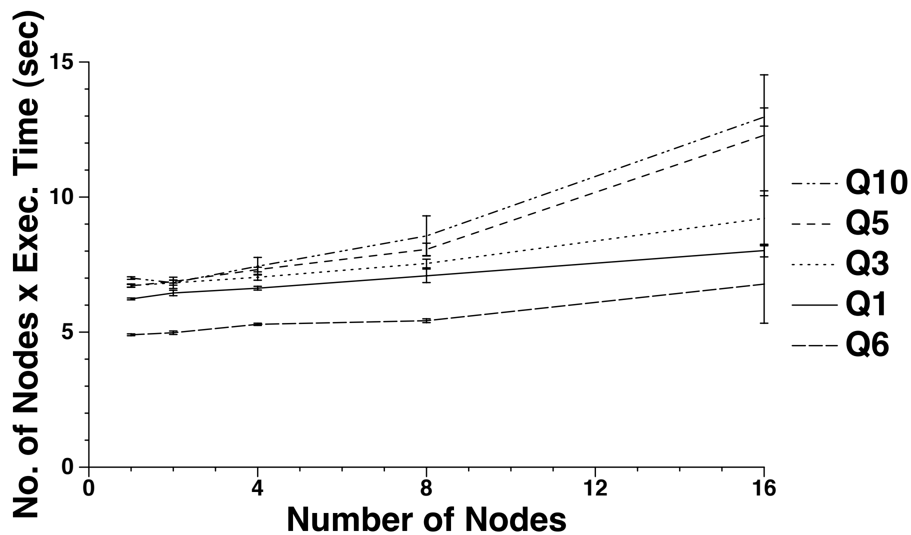


Figure 4.12: Normalized running time: TPC-H Scale Factor 0.5, 1-16 nodes. The normalized time shown, the product of execution time and the number of nodes, would be flat if the system scaled perfectly. All queries scale somewhat well, though Q10 and Q5 rehash more data and therefore scale less data than the others.

Scaling Nodes

Figure 4.9 shows execution times for STBenchmark (at 800,000 tuples/relation) for 1 to 16 physical nodes, while Figure 4.10 shows times for TPC-H queries over the 500MB data set (scale factor 0.5). Ideally, the running times would be halved each time we double the number of nodes. Our results come very close to matching this expectation for all of the TPC-H queries and about half of the STBenchmark queries. In the other STBenchmark queries (in particular Copy), so much data is returned (because the tuples consist of many long strings), that collecting the results at the query initiator becomes a bottleneck. With 16 nodes, all but 0.1 sec of the Copy query is spent transmitting and receiving the results. We conducted separate experiments to verify that performance is mostly limited by network bandwidth, with some additional performance degradation due to the unmarshalling and storage at the query initiator. All queries continue to show some performance improvement as the number of processing nodes increases.

If scaling were perfect, the product of the number of nodes and the execution time, which we term the *normalized running time*, would be constant. We present normalized running times for the experiment just described in Figures 4.11 and 4.12. As one might expect, there is overhead in scaling to larger numbers of nodes, in addition to the benefit of increased parallelism demonstrated above; the lines in these figures are not constant. However, they are not steep, showing that the overhead is not too great. Also as one might expect, the queries that exchange data between nodes to perform a join (Q3, Q5 and Q10 for TPC-H, and Join and Correspondence for STBenchmark) have more overhead, as exchanging data is a many-to-many operation that becomes more complex as the number of nodes increases; more network connections are used, and the routing table consulted to determine where tuples should be sent becomes larger. It is important to remember, though, that as shown in Figures 4.9 and 4.10, there is still benefit to increasing the number of nodes.

Figures 4.13 and 4.14 show the total network traffic while executing these queries, and Figures 4.15 and 4.16 show the per-node traffic. As expected, the network traffic increases as we scale up the number of nodes, but not dramatically so, and the per-node traffic (after rising significantly when we move from single-node computation to distributed operation) continues to decrease as nodes are added to the system.

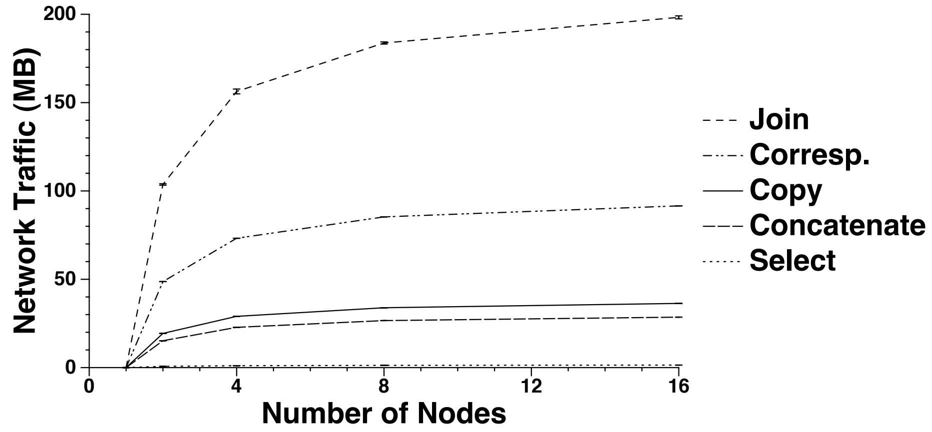


Figure 4.13: Network traffic: STBenchmark, 800K tuples/relation, 1-16 nodes. Network traffic increases as the number of nodes increases, but plateaus.

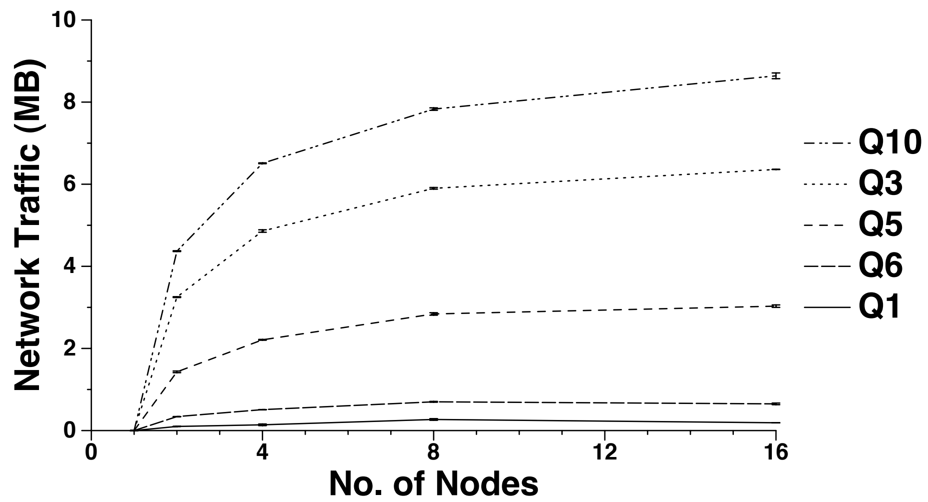


Figure 4.14: Network traffic: TPC-H Scale Factor 0.5, 1-16 nodes. Network traffic increases as the number of nodes increases, but plateaus.

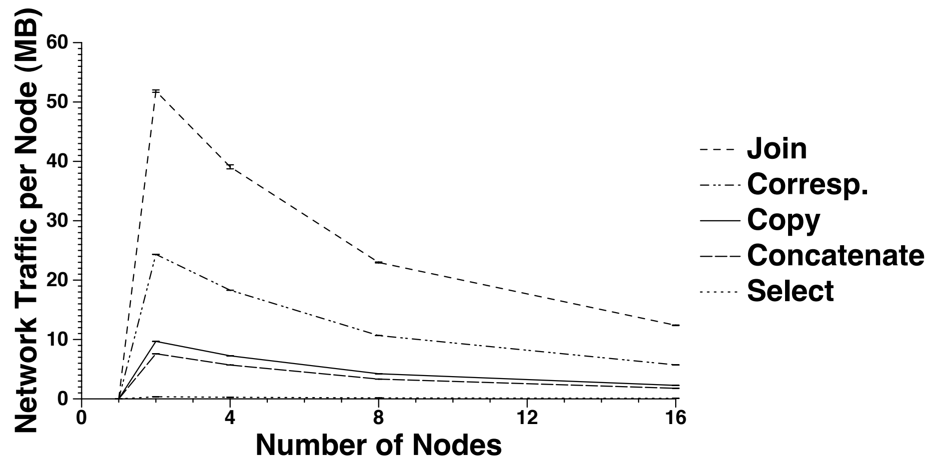


Figure 4.15: Per-node network traffic: STBenchmark, 800K tuples/relation, 1-16 nodes. Per-node network traffic continues to decrease as the number of nodes increases.

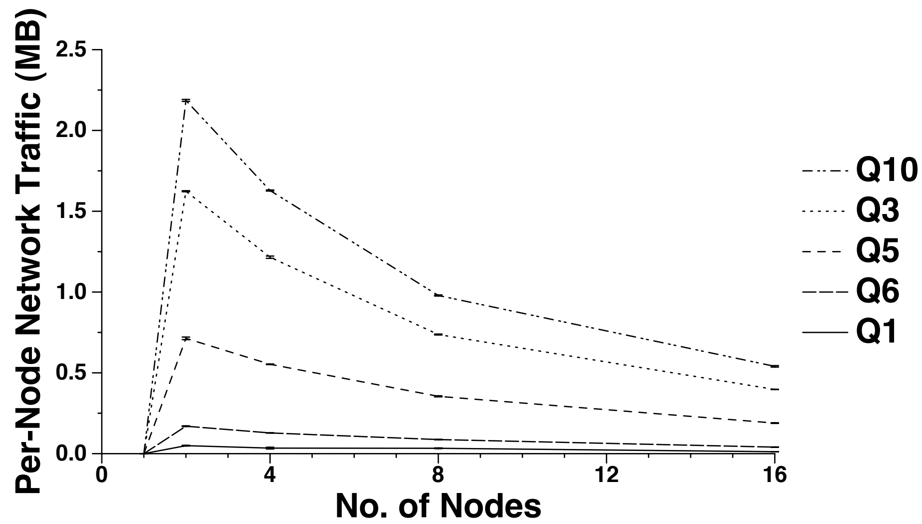


Figure 4.16: Per-node network traffic: TPC-H scale factor 0.5, 1-16 nodes. Per-node network traffic continues to decrease as the number of nodes increases.

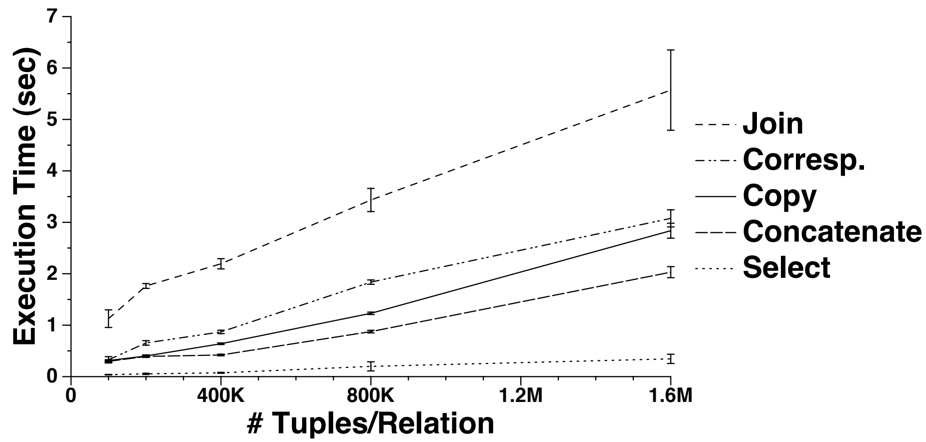


Figure 4.17: Running time vs. data size, STBenchmark, 8 nodes. For selection and foreign-key joins, performance is linear in the amount of data, as expected.

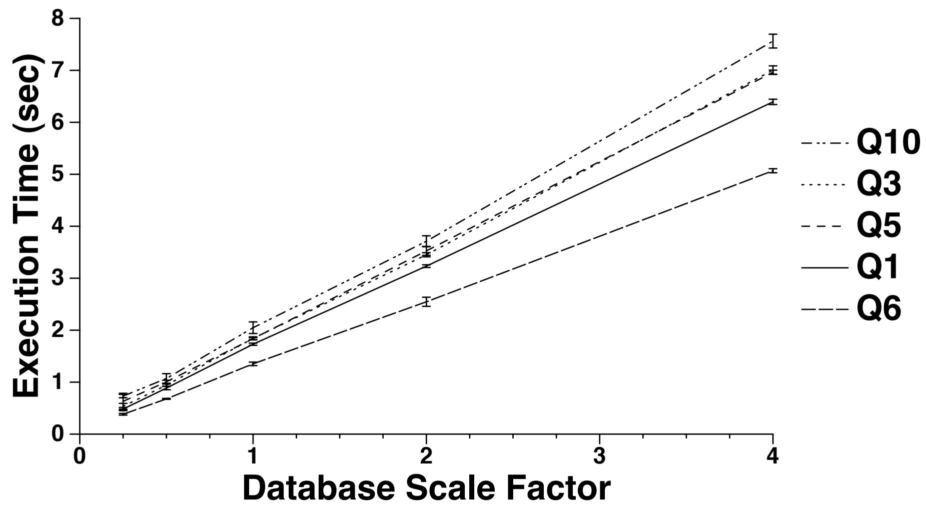


Figure 4.18: Running time vs. data size, TPC-H, 8 nodes. For selection and foreign-key joins followed by aggregation, performance is linear in the amount of data, as expected.

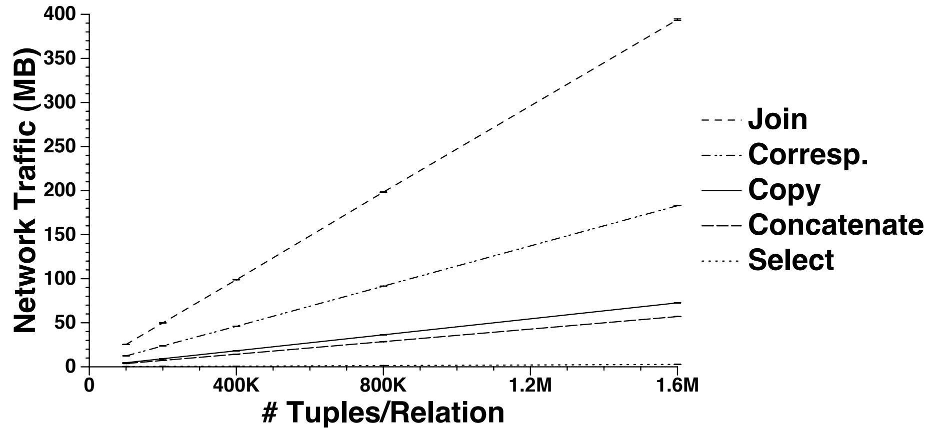


Figure 4.19: Network traffic vs. data size, STBenchmark, 8 nodes. For selection and foreign-key joins, network traffic is linear in the amount of data, as expected.

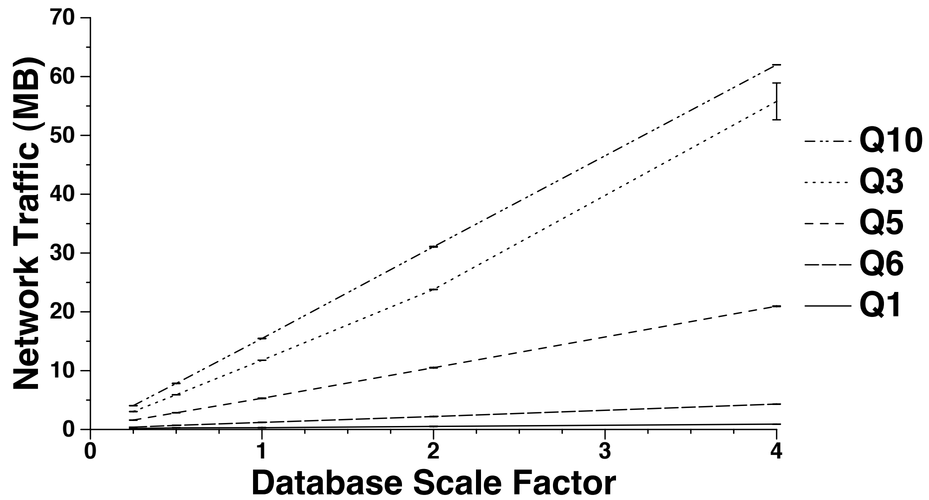


Figure 4.20: Network traffic vs. data size, TPC-H, 8 nodes. For selection and foreign-key joins followed by aggregation, network traffic is linear in the amount of data, as expected.

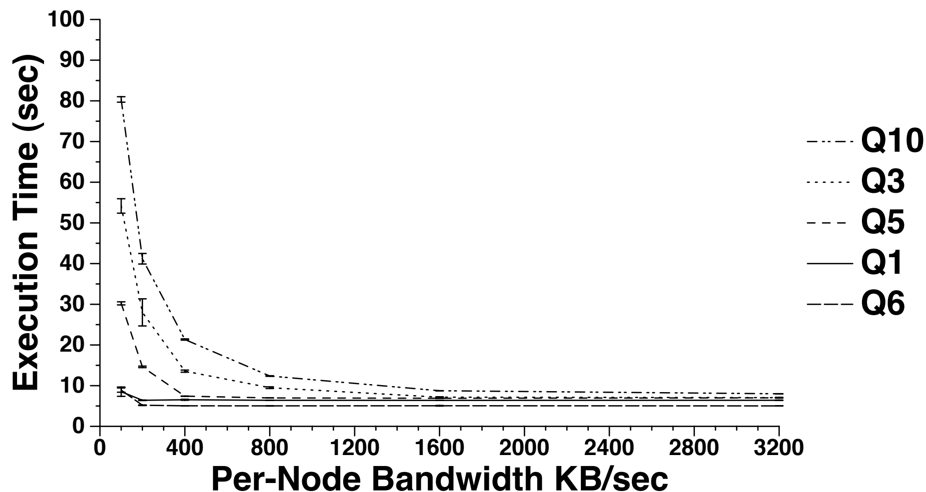


Figure 4.21: Running time vs. per-node bandwidth, 8 nodes, TPC-H scale factor 4. Performance is severely constrained by available bandwidth only at lower bandwidths. At connection speeds reasonable for a corporate or academic network, bandwidth is not a limiting factor.

Scaling Data Set Size

We next consider the effects of scaling the data. Figure 4.17 shows execution times for STBenchmark on the 16-node cluster for 100K to 1.6M tuples/relation, and Figure 4.18 shows the same for the TPC-H queries over the 8-node cluster while varying the data size from 250MB to 4GB (scale factors 0.25 to 4). Figures 4.19 and Figure 4.20 show total network traffic for the same scenarios. Execution times and network traffic for all queries scale approximately linearly in the size of the data, as one would expect since there are only foreign-key joins and the data is fairly evenly distributed. We conclude that our system scales well on a LAN, and move on to consider other network settings.

Performance over a Simulated Wide Area Network

We next consider possible variations on Internet connectivity among compute nodes. We made use of the traffic shaping and network emulation features built into recent versions of Linux to simulate various parameter changes. Specifically, we used NetEm to delay outgoing packets, simulating a higher latency network, and we used the HTB queue discipline to simulate a lower bandwidth network. Here we focus on the TPC-H benchmark, since STBenchmark, due to its large strings, becomes increasingly bandwidth-constrained at the query initiator, and since we feel its data is actually less representative than TPC-H's.

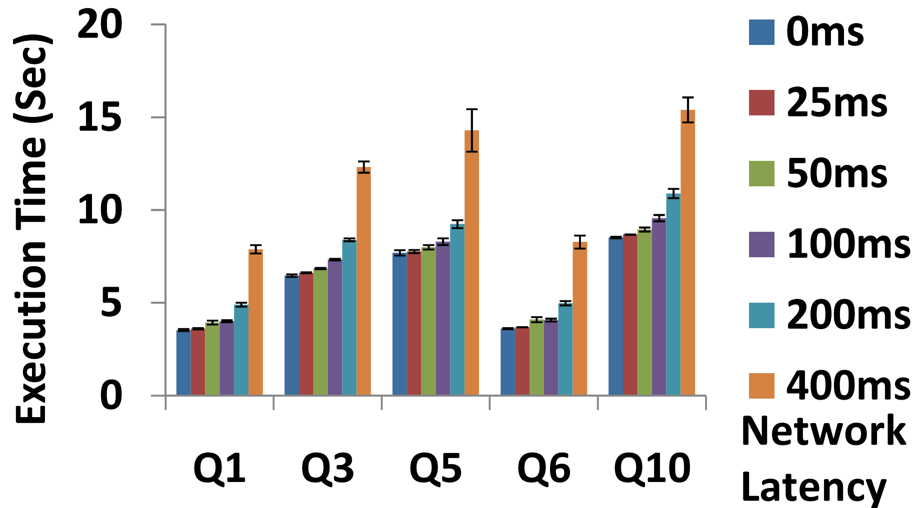


Figure 4.22: Running time vs. added network latency, 16 nodes, TPC-H scale factor 1. Only extremely high latency has a significant impact on query performance.

Limited Bandwidth Settings.

Our experimental results, shown in Figure 4.21, demonstrate that while performance suffers in very low-bandwidth connections, execution times are degraded but reasonable for the bandwidths likely to be available between academic, institutional, or corporate users ($>$ 400 kB/sec). Queries 1 and 6, which perform no rehash operations and therefore send much less data over the network, are less impacted than queries 3, 5, and 10, which join multiple relations and rehash data while doing so.

Higher Latency Settings.

We also performed a series of experiments to determine the effects of added latency on query execution performance. For these experiments, we used 16 cluster nodes and TPC-H scale factor 1. Since the nodes are on a high-speed LAN, the latency of the physical network is $\ll 1$ ms, so the added latency is by far the dominant factor in the experiments. Figure 4.22 shows the effect of increasing latency uniformly on query performance. The effect of realistic ($\leq$ 200 ms) latencies was minimal. Figure 4.23 shows the effects of adding high latency at some nodes. The query initiator is always the last node to be slowed, which explains the large increase from 15 nodes to 16 nodes. In general, the system is only sensitive to higher latency at the query initiator or if many of the nodes are high latency; a single high latency node has very limited effects on performance. Figure 4.24 shows the

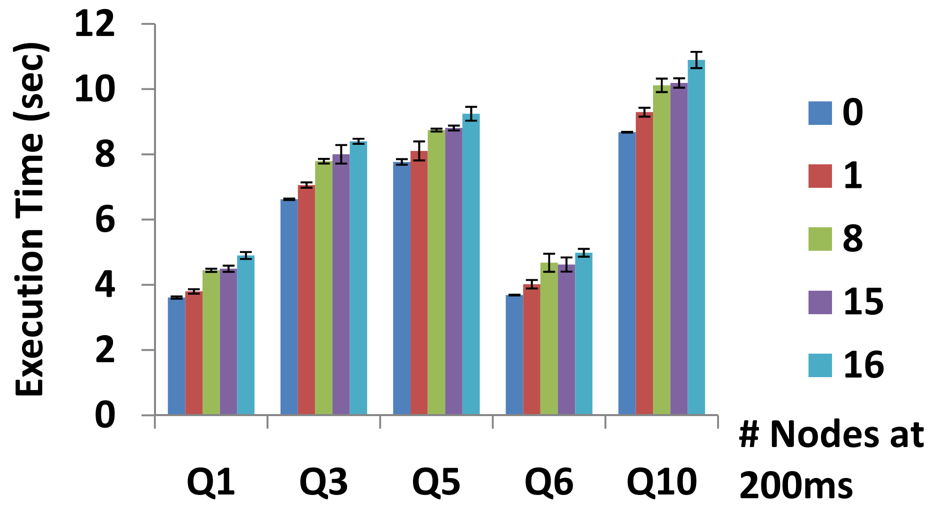


Figure 4.23: Running time vs. number of higher-latency nodes, 16 nodes, TPC-H scale factor 1. The effect of latency on performance is most strongly correlated with the highest latency node.

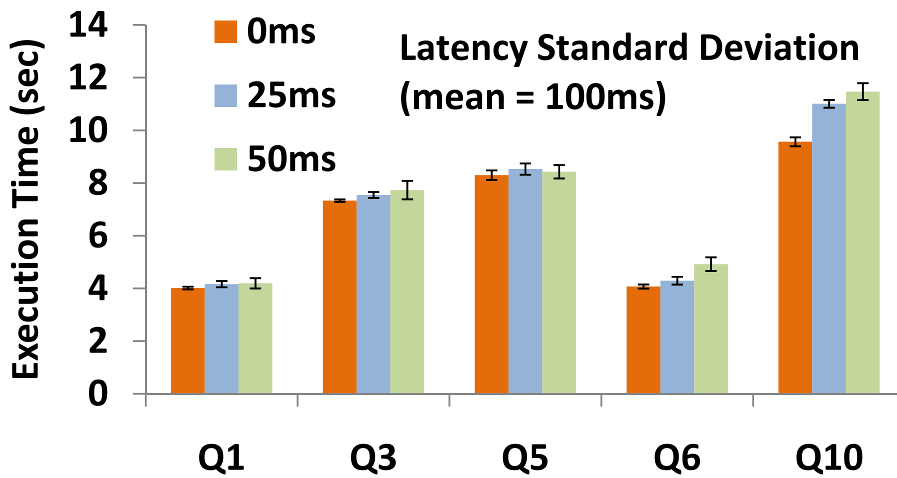


Figure 4.24: Running time vs. latency variability for normally distributed latencies with mean 100ms, 16 nodes, TPC-H scale factor 1. The latency variance has a limited influence on performance.

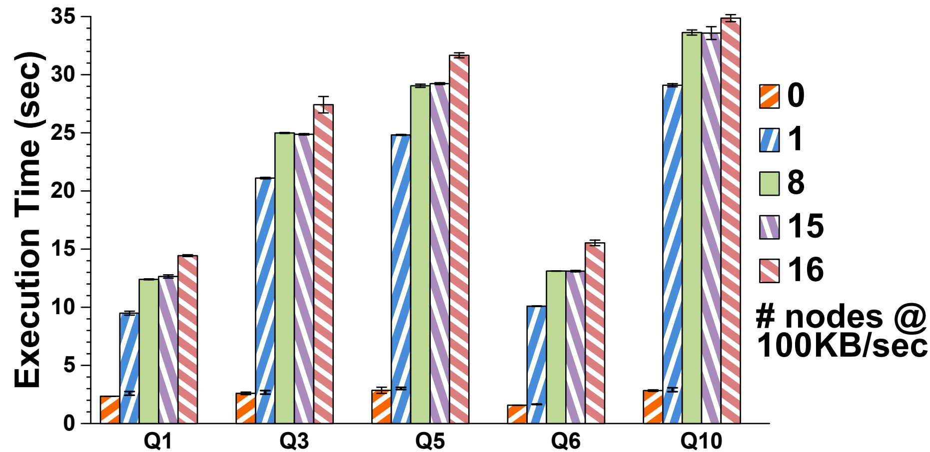


Figure 4.25: Running time vs. number of slow nodes running at 100 KB/sec, 16 nodes, TPC-H scale factor 1. Fast nodes run at 1600 KB/sec. This experiment validates that the slowest node is the dominant factor in determining running time. Marks on the 1 node bars indicate the performance of a modified routing table that assigns 1⁄16 as much of the key space to the slow node as to all other nodes; this shows that balancing the routing table by available bandwidth compensates for slower nodes quite effectively.

effect of non-constant added latency on system performance. Here the latency of each outgoing IP packet was normally distributed, with a mean of 100 milliseconds. The figure shows the effect of changing the variance of the added latency. As before, the system is resilient to unstable conditions; increasing the variability caused only a small decrease in query performance.

Validation that Worst-Case Governs Performance

In our optimizer, we assumed that *worst-case* CPU and network performance governs system performance. To determine if this is correct, we ran the following experiment. While keeping the remaining nodes at 1600 kB/sec, we limited some of the nodes to 100 kB/sec and measured query performance; the last node to be reduced to the slower bandwidth was the query initiator. Figure 4.25 shows that having only one node limited to the slower transmission rate greatly reduces query performance; from there on the effect is not very pronounced. This confirms our approximation that the slowest node will limit overall performance.

It seemed likely that we could improve performance, at least for a few slower nodes, by adjusting the ranges in the DHT key space so that the slower nodes were responsible for less data. We therefore modified the routing table for the case with one slow node so that the slow node is responsible

for $1/256$ of the key space ($1/16$ of what it was before), and each of the remaining fifteen nodes for $17/256$ of the key space, or $17/16$ of what it was before. The results are the single data points overlaid on Figure 4.25. Encouragingly, the execution times for this "weighted" scenario come very close to those for the case where all of the nodes are fast. Performing such tuning in an automatic fashion is a therefore a promising avenue for future research, and we propose such work in Chapter 5.

Scalability to Larger Numbers of Nodes

Since we have a limited number of local machines in our cluster, we next tried several alternatives to scale to higher numbers. Our initial efforts were with the PlanetLab network testbed — but disappointingly, we found that most nodes here were severely underpowered and overloaded, and disk- and memory-intensive tasks like ours were constantly thrashing, resulting in inconsistent and uninformative results.

Instead, we leased virtual nodes from Amazon's EC2 service — something we envision ORCHESTRA's user base doing as needed. Amazon has data centers geographically distributed across the world, so round-trip times are short and bandwidth is high. We used EC2's "large" instances with 7.5GB RAM, and a virtualized dual-core 2GHz Opteron CPU; this is more memory than is needed, but the next smaller EC2 instance size had too little RAM. We experimented with the TPC-H scenario, as performance on STBenchmark at the data sizes we could generate was either too fast to be measured reliably or dominated by the cost of collecting the results at the query initiator.

We varied the number of total participants in the setting from 10 to 100, using TPC-H scale factor 10 (10GB data). Network traffic results, shown in Figures 4.26 and 4.27, are similar to the results shown in Figures 4.14 and 4.16 for smaller numbers of nodes. Execution times are shown in Figure 4.28. As before, increasing the number of nodes leads to a dramatic decrease in execution time. Figure 4.29 shows normalized execution times, again by considering the product of the number of nodes and the execution time; if our system scaled perfectly, the line would be totally flat. We see that some queries perform significantly faster (on a normalized basis) as the number of nodes increases from ten to forty or so. This is because a greater fraction of the TPC-H relations can be cached in memory; observe that Q1 and Q6, which read data only from the *Lineitem* relation (which fits into the cache at all numbers of nodes) do exhibit this behavior. After this, the per-node overhead increases, and we see less than perfect scaling as we move from forty to 100 nodes; however, the scaling is still

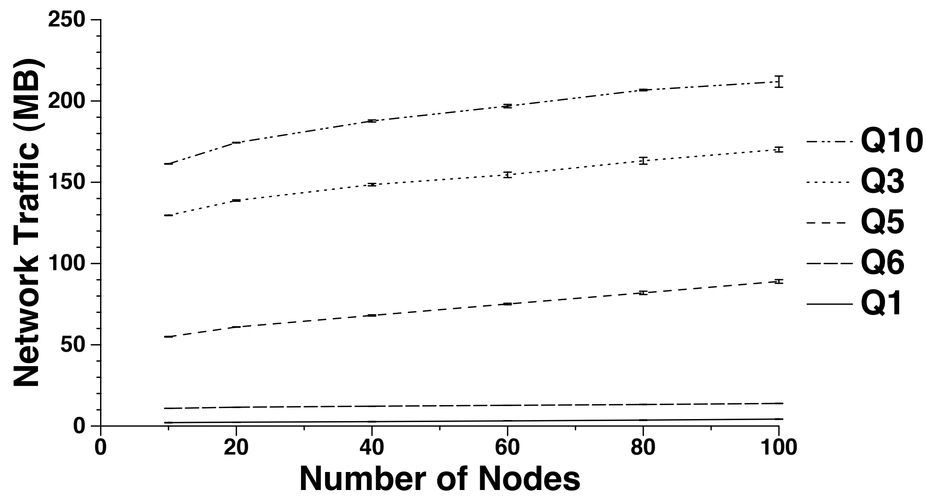


Figure 4.26: Total traffic on 10 to 100 EC2 nodes, TPC-H scale factor 10. Overall network traffic increases only gradually as the number of node increases.

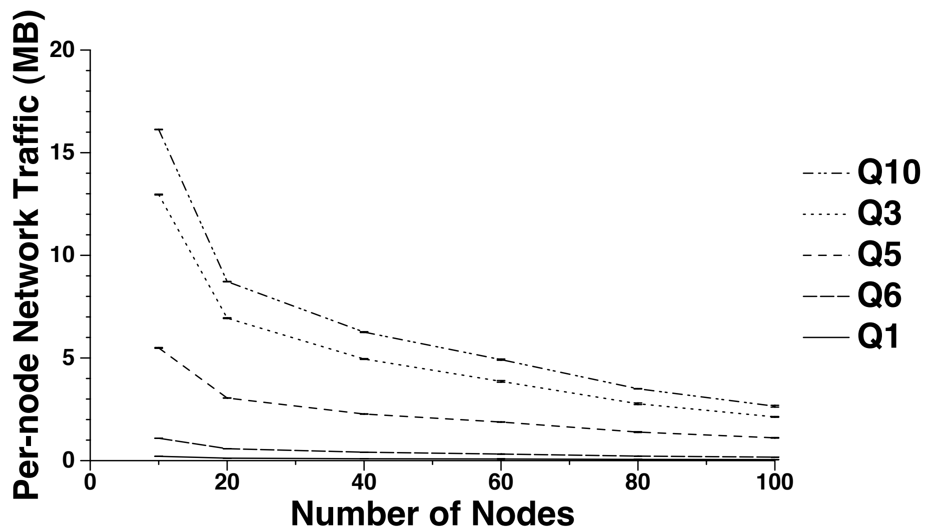


Figure 4.27: Per-node traffic on 10 to 100 EC2 nodes, TPC-H scale factor 10. Per-node traffic continues to drop as the number of nodes grows.

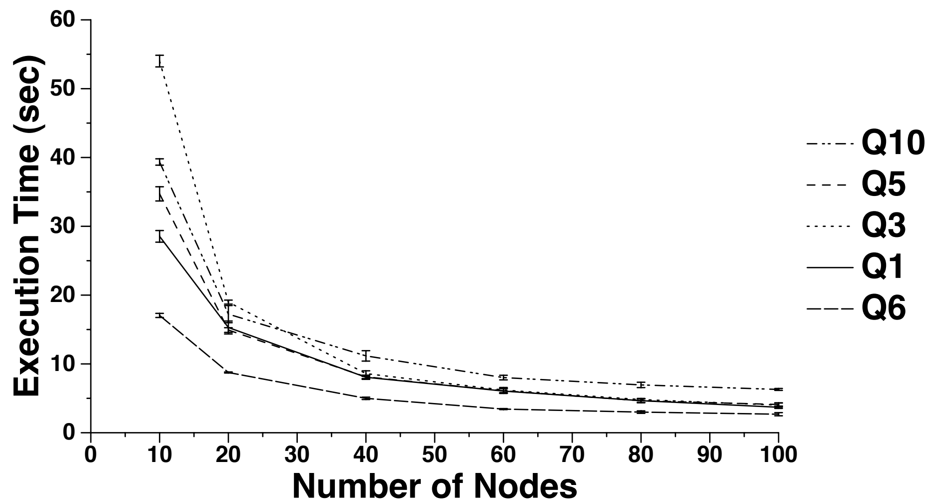


Figure 4.28: Execution time on 10 to 100 EC2 nodes, TPC-H scale factor 10. Adding more nodes continues to improve execution time.

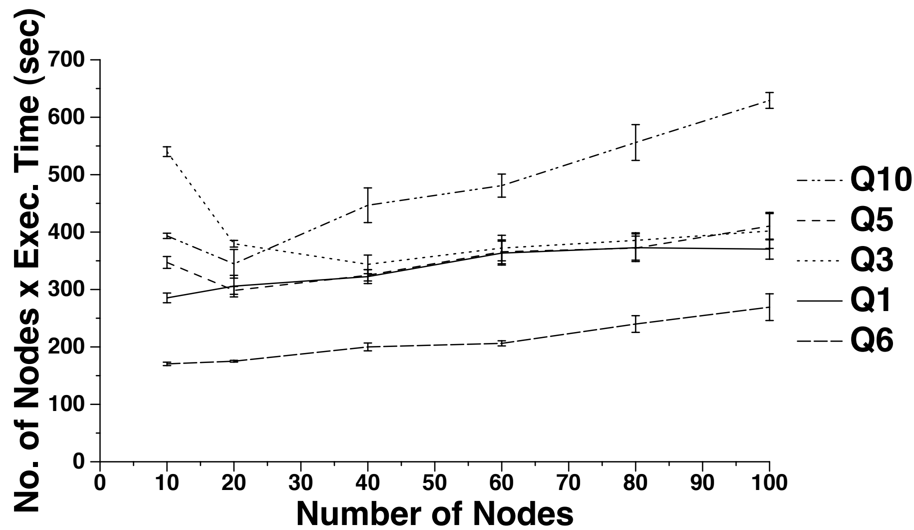


Figure 4.29: Normalized execution time on 10 to 100 EC2 nodes, TPC-H scale factor 10. The normalized time shown, the product of execution time and the number of nodes, would be flat if the system scaled perfectly. While the system does not scale perfectly, the normalized execution times increase slowly relative to the increase in the number of nodes, due to routing and distribution overhead. The initial decrease in normalized execution time for certain queries is due to the entire working set of tables for those queries fitting into the nodes' caches.

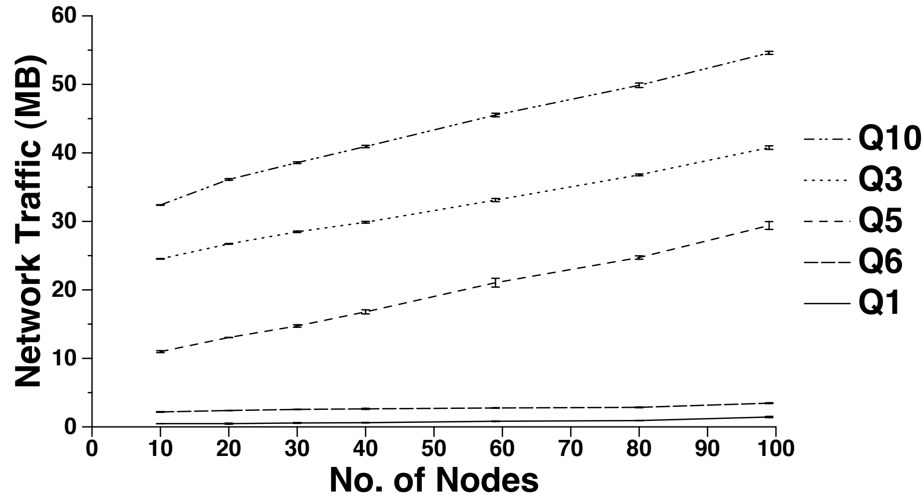


Figure 4.30: Total traffic on a mix of EC2 and cluster nodes, TPC-H scale factor 2. Network traffic increases slowly as we increase the system size.

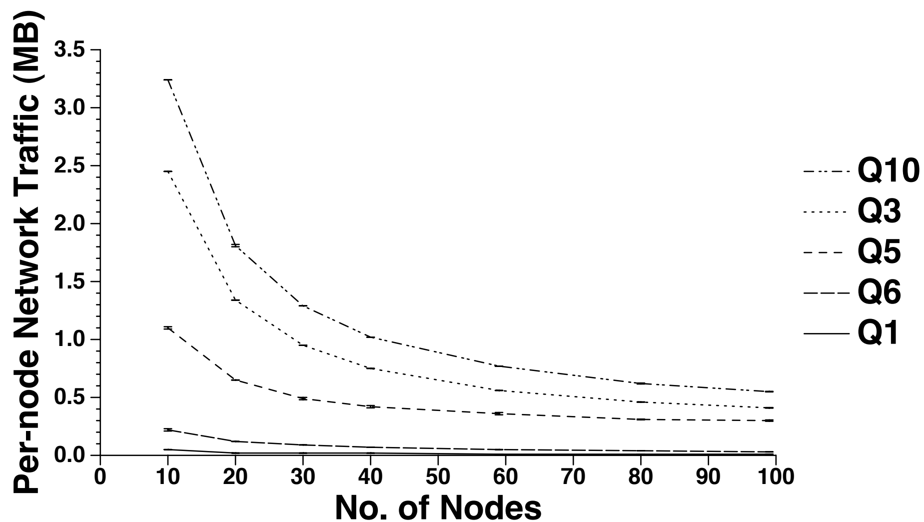


Figure 4.31: Per-node traffic on a mix of EC2 and cluster nodes, TPC-H scale factor 2. Per-node traffic continues to fall as we increase the system size.

good, as shown by the fact that running times continue to decrease in Figure 4.28, and by the slope of the lines in Figure 4.29. This experiment validates the scalability of our system to large numbers of nodes.

We also experimented with a mix of EC2 nodes and machines from our cluster. For this experiment, we employed EC2's "small" instances with 1.7GB RAM, and a virtualized CPU that is nom-

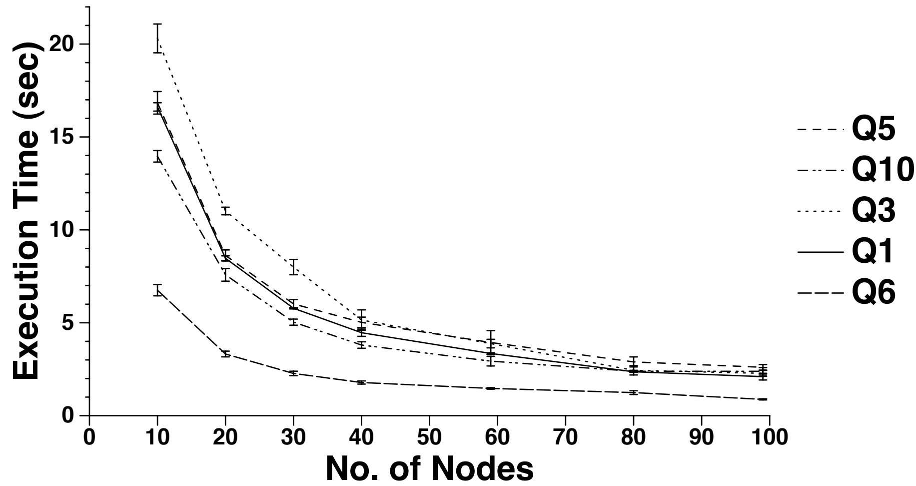


Figure 4.32: Execution time on a mix of EC2 and cluster nodes, TPC-H scale factor 2. Increasing system size continues to result in improved performance.

inally a 2.5 GHz Opteron but varies in performance. For smaller numbers of nodes (up to 30), we used an equal number of cluster nodes and EC2 nodes; for larger numbers, we used 30 cluster nodes and the rest came from EC2. To accommodate the relatively small about of RAM in the EC2 nodes, we used TPC-H scale factor 2. Total and per-node network traffic is shown in Figures 4.30 and 4.31, and execution time in Figure 4.32. The trends are very similar to to trends for experiments using exclusively EC2 nodes, shown in Figures 4.26, 4.27, and 4.28. These results confirm the ability of our system to scale to large numbers of nodes in a real-world setting with geographically diverse nodes.

Failure and Recomputation

Finally, we study recovery when a node fails or becomes unreachable. One option is to abort the query and restart it over the remaining nodes. The other is to use the remaining nodes to recompute the "lost" results. Our experiments used 8 nodes and TPC-H scale factor 2.

Incremental Recomputation versus Total Restart

To explore the trade-offs between incremental recomputation versus full restart, we first ran a series of experiments using Q1 (a selection and aggregation query) and Q10 (which performs three joins followed by an aggregation), chosen to represent the two classes of TPC queries we studied. We started each query and at varying points after the start of the query (before it finished) we caused one of the

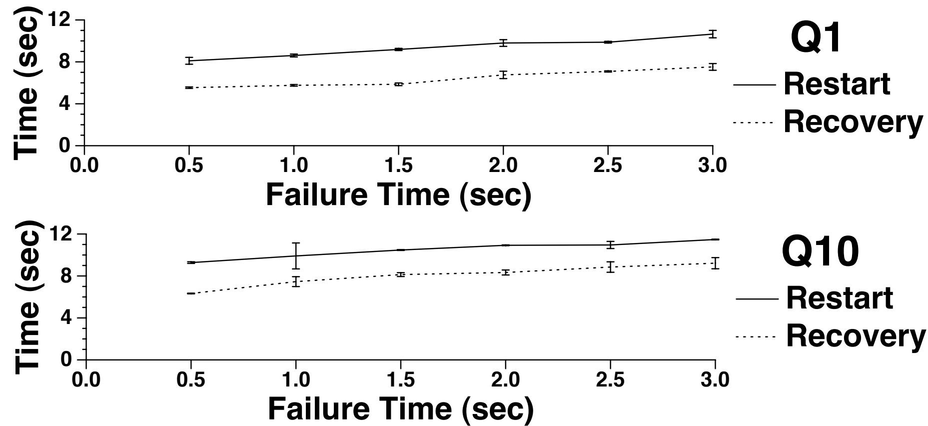


Figure 4.33: Running times for Q1 and Q10 with a failure with and without incremental recovery, 8 nodes, TPC-H scale factor 2. The failure occurred after the number of seconds shown on the *x* axis. As shown, regardless of when the failure occurred, incremental recovery is significantly faster than restart.

nodes to fail. To avoid giving incremental recomputation an unfair advantage, we recompute using the same routing tables (which spreads the range of the failed node evenly over the nodes holding its replicated data). Figure 4.33 shows performance results for Q1 and Q10. In both cases, incremental recovery outperforms aborting and restarting by approximately 20%, validating the approach. Execution is slow for both techniques (compared to no failure) due to the cache misses inherent when a new node takes over a portion of the substrate key space.

Overhead of Incremental Recomputation

Incremental recomputation requires more data to be stored and sent over the network (to track the provenance of intermediate results), and requires that all intermediate results be kept around until the end of the query. Clearly, if this adds significant overhead to an average query, it may actually be preferable to restart after nodes fail. We measured the overhead of incremental recovery support on the TPC-H queries, which we briefly summarize here. Execution time overhead for each query is shown in Figure 4.34, while network traffic overhead is shown in Figure 4.35. As expected, recovery support slightly increased execution time: queries ran from 2%-7% slower. Network traffic increased by negligible amounts, at most 2% (for Q10). In our view, this overhead is low enough to make it worthwhile if there is a reasonable expectation of node failure — particularly for long-running queries where the cost of restart may be high. Such an expectation goes up as more nodes

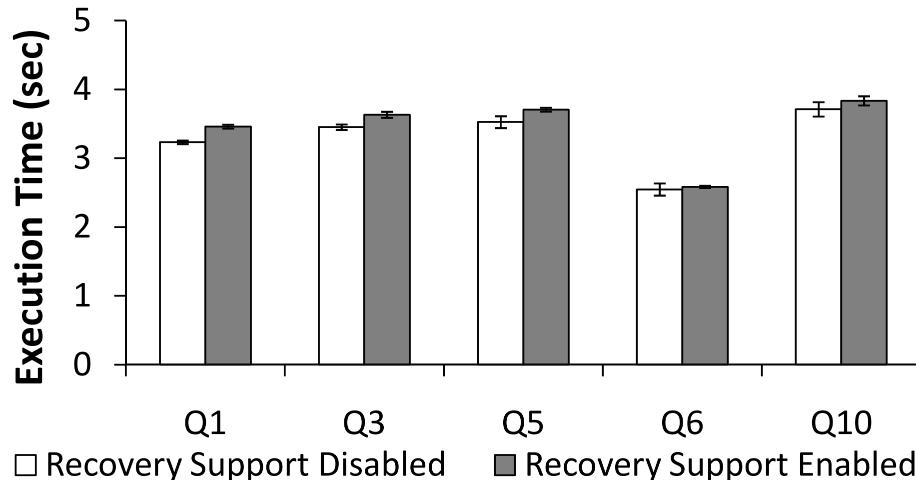


Figure 4.34: Execution time overhead of incremental computation, TPC-H scale factor 2, 8 nodes. Adding support for incremental recovery results in a small increase in execution time if there is no failure, mainly due to the cost of maintaining the per-tuple annotations holding the contributing node sets.

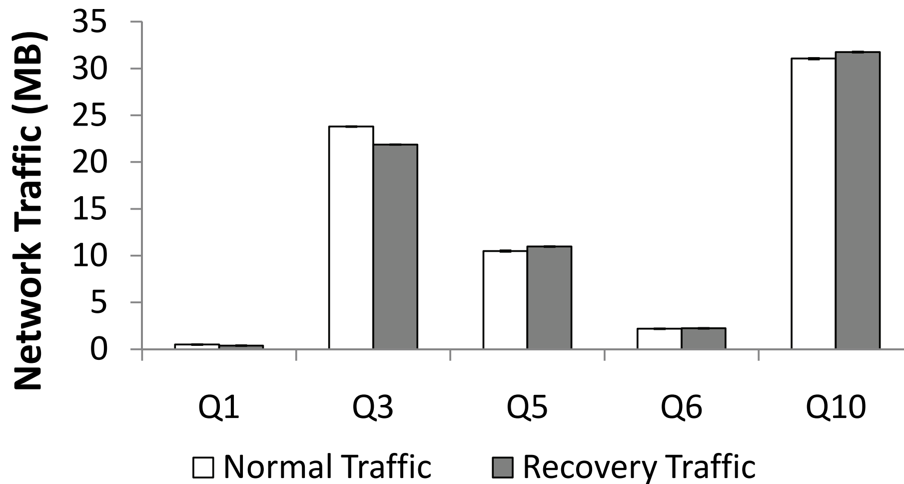


Figure 4.35: Network traffic overhead of incremental recovery, TPC-H scale factor 2, 8 nodes. Adding the per-tuple annotations needed for incremental recovery results in only a change in the amount of network traffic. We attribute the *decrease* in traffic for Q3 to experimental error arising from the effects of thread scheduling and buffering prior to transmitting data across the network.

join (and query running times go down, reducing the overall amount of overhead). Also, if query performance is limited by available network bandwidth, incremental recovery becomes almost free due to the low network overhead, and restarting becomes more expensive. Finally, we also feel that in many situations *predictable query performance* is more important that average query performance. If we can significantly reduce the worst-case running time at a small cost to the average case, this may lead to better perceived performance, especially in a user interactive system.

4.6 Conclusions and Analysis

This chapter has shown how to provide a reliable peer-to-peer storage and query execution engine for a CDSS. This involves a richer networking substrate, novel differential indexing schemes to guarantee the correct versions of all tuples are used during processing, and a query evaluator that is carefully matched to this substrate. We developed techniques for handling failures through incremental or full recomputation, and showed the trade-offs between these approaches. We experimentally validated the performance of our approach in a variety of settings. We experimentally demonstrated the need for load balancing in heterogeneous peer-to-peer networks, and validated a potential solution by manually altering the key space partitioning to favor more powerful nodes. We will explore automatic means of performing such load balancing, and address short comings of the totally even partitioning scheme we used here, as part of a more general approach in Chapter 5.

Chapter 5

Load Balancing

Real peer-to-peer and cloud systems contain many nodes of varying capabilities, varying background loads, or both. Some are better connected to the Internet, some are more computationally powerful, some have faster disks, some have more memory, some also perform other tasks, and so on. The peer-to-peer query processor for ORCHESTRA presented in Chapter 4 did not take this into account. It implicitly assumed that all nodes were equally powerful when it assigned equal portions of the distributed hash table key space to each node in the range partitioning scheme described in Section 4.2; this ensured that no node was assigned more than its "fair share" of the data, assuming it is uniformly distributed throughout the key space. This provided optimal performance in the case that the nodes were equally powerful and performing no other tasks, which was the scenario used in the experimental evaluation in Section 4.5.

Unfortunately, this scenario is not likely to occur in practice. In a wide-scale distributed system, differences in network connectivity are more or less inevitable, and hardware heterogeneity is also to be expected. Especially in the case of an *ad hoc* cloud of user-supplied nodes, background load due to existing use of hardware resources is also very likely. An experiment described in Figure 4.25 of Section 4.5 showed that the least powerful (in this case, lowest bandwidth) node was a limiting factor in query processing performance. This figure also shows the result of a preliminary experiment, in which the amount of the key space assigned to each node is weighted by its available bandwidth. With this tweak, the performance of a system with one slow node was very close to the default evenly balanced performance when all nodes were at the same faster speed. This promising result is part of the motivation for this chapter: intuitively, assigning more of the key space to more capable nodes

will result in better performance, solving the problem of *node heterogeneity*.

Additionally, the totally even partitioning used in the previous chapter violated several traditional properties of peer-to-peer networks. In traditional DHTs like Pastry Rowstron and Druschel (2001) and Chord Stoica et al. (2001), a node arrival or departure only changes the key space partitions owned by nodes that are "nearby" in the key space to the ID of arriving or departing node. In the even partitioning proposed in Section 4.2, *any* node arrival or departure will cause some change to *all* other nodes; in smaller networks, the amount of change can be very large. We would like to explore ways to retain Pastry and Chord's *partitioning resiliency* property while also ensuring the partitioning has other desirable properties, like evenness or being skewed towards more capable nodes.

A third shortcoming of the even range partitioning was that it didn't account for data skew, and the resulting load skew. Hashing with a sufficiently good hash function does a nice job of compensating for range-based skew, or various other kinds of skew (say a bias towards even numbered IDs, or names that end in an 's,' or many other properties). However, it cannot compensate for large numbers of data items with the same DHT ID; this is possible in our system, and perhaps even likely, if a relation has a compound primary key. This situation leads to a non-uniform data item density in the DHT key space, meaning that even range partitioning may still lead to a non-even partitioning of data items. We would like to explore ways to compensate for this issue of *data skew* as we solve the aforementioned problems of node heterogeneity and partitioning resiliency.

These three problems with partitioning motivated us to entirely rethink the partitioning scheme used in Chapter 4. In this chapter, we replace that partitioning scheme (while leaving the rest of the system unchanged) with a new approach to partitioning that exploits the *replicated* property of the data in the system. Since each data item is stored at multiple nodes, the system has some flexibility at query execution time as to how to allocate nodes to regions of the DHT key space. If we exploit this flexibility in a sufficiently clever fashion, we can find a unified solution to these three problems.

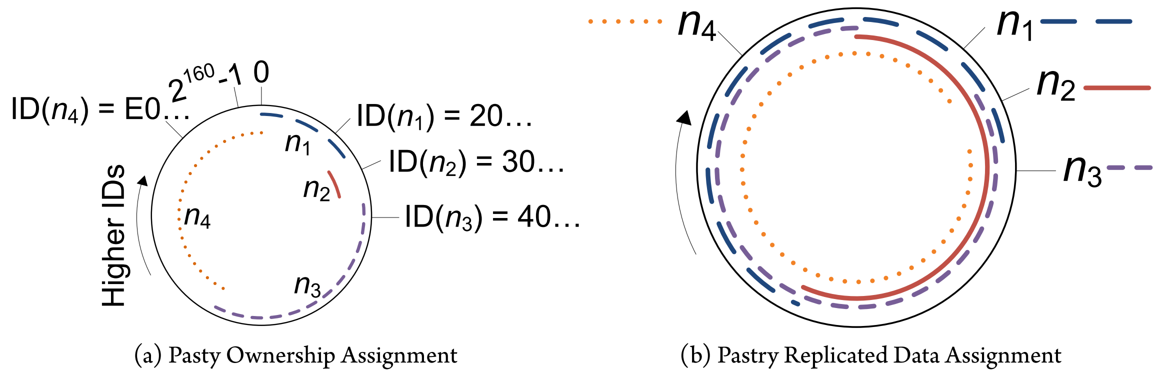
This chapter is structured as follows:

- Section 5.1 describes in detail how replication affords us partitioning flexibility.
- Section 5.2 begins with an introduction of constrained optimization as a class of problems, and then states the operations a constraint solver must support in order for our approach to work.
- Sections 5.2 and 5.2 show how we cast partitioning optimization as a constrained optimization

problem, and consider both node heterogeneity and data skew.

- Section 5.2 gives a concrete example of the type of constrained optimization programs generated for input into the Minion constraint solver.
- Section 5.3 explores the amount of imbalance present in the base partitionings created by Pastry-style partitioning, and shows how much our approach can reduce that under a variety of settings.
- Section 5.3 explores how performance of the ORCHESTRA query processor on a variety of OLAP and schema mapping queries is affected by partitioning optimization, and shows the effect of partitioning optimization on data skew.
- Section 5.3 compares performance of ORCHESTRA with several traditional RDBMSs.

While we defer a full discussion of related work to Section 6.4, we briefly discuss here the reason that the standard techniques for load balancing from the literature do not apply. The canonical approach to load balancing in distributed hash tables is that of *virtual servers*, introduced in the context of Chord's distributed file system CFS (Dabek et al., 2001). In this approach, each physical node in the system runs many virtual nodes, each with its own node ID and own partition of the DHT key space. If there are enough virtual nodes at each physical node, with high probability each physical node will have an approximately even share of the key space assigned to it by all of its virtual nodes. More powerful physical nodes can be made more likely to receive a large fraction of the key space by running more virtual servers. The approach breaks down in our context because of the locality assumptions (described in Section 4.3) between index pages and the tuples they reference; preliminary experiments showed that performance using various numbers of virtual nodes at each physical node for a 16-node system offered performance so dramatically worse than the even partitioning used in the previous chapter that we did not study it further. Our system stores index pages near referenced tuples in the DHT key space, making it likely that the index pages are never sent over the network. If we assign each node many smaller discontiguous regions of the key space, more of the index pages are sent over the network, and performance suffers. Therefore, we need a balancing solution that assigns each node a contiguous region of the key space, to minimize this source of network overhead.



(a) Pasty Ownership Assignment (b) Pastry Replicated Data Assignment

Figure 5.1: Partitioning of the DHT key space. The data is replicated with replication factor $r = 3$.

5.1 Partitioning Flexibility

Recall that Pastry-style partitioning uses a circular DHT key space (from 0 to $2^{160} - 1$) which wraps around. Each node has an ID in the DHT key space, typically the hash of it's IP address (or IP address and some unique local identifier, such as TCP port, if multiple virtual nodes are being run on the same physical node). These nodes are then placed onto the DHT key space ring, and each node is responsible for, or "owns," the region of the key space closer to it than to its neighbors. This is shown in Figure 5.1a. If data is then replicated with a replication factor r, each data item (and therefore each portion of the key space) is stored at $r - 1$ other nodes as well, for a total of r copies. In this chapter, we assume that r is always odd, and that each data item is replicated at $\frac{r-1}{2}$ nodes counterclockwise from the node that owns it, and $\frac{r-1}{2}$ node clockwise as well. This leads to the overlapping assignment of regions of the DHT key space (for purposes of data storage) to nodes as shown in Figure 5.1b.

In a DHT-based storage system and query processor such as ours, the partitioning of the key space is used both to determine where persistent data is retrieved from persistent storage, and how intermediate results are partitioned. Each region of the key space must be assigned to exactly one node. One such assignment in shown in Figure 5.1a. Note, however, that this assignment is rather lopsided.

Definition 9 (Routing Imbalance). *We use the* routing imbalance *metric as a measure of the quality of a partitioning. The routing imbalance is the largest fraction of the key space assigned to one node divided by the fraction assigned to each node in a perfectly even assignment; this is equivalent to the largest fraction times* n*, the total number of nodes. Since in any assignment one node must be responsible for at least* $\frac{1}{n}$ *of the key*

space, and can be responsible for at most all of it, the routing imbalance ranges from one to n. If all nodes in a system are equally powerful, the node that is assigned to the largest fraction of the key space, and therefore has more work routed to it, is typically the bottleneck; therefore higher routing imbalance is strongly correlated with the execution time of a query. For the time being we assume homogeneous nodes, and will address node heterogeneity shortly.

We can use the routing imbalance to quantify how lopsided the partitioning in Figure 5.1a is. From the node IDs, for Pastry-style partitioning we can determine that n_1 owns the region `[0x00…, 0x25…)`, n_2 the region `[0x25…, 0x35…)`, n_3 the region `[0x35…, 0x90…)`, and n_4 the region `[0x90…, 0x00…)`. n_4 therefore owns $\frac{7}{16} \approx 0.44$ of the key space, giving a routing imbalance of 1.75. If the nodes are equally powerful, this partitioning would lead to much worse performance than the totally even partitioning we used in Chapter 4, since n_4 in particular would be overloaded, and n_2 in particular would be underloaded.

Since we have committed to preserving the property of partitioning resiliency, we cannot directly apply the techniques of even partitioning from Section 4.2 for all operations. Scans of persistent storage must occur where a copy of the data is stored. We could, in principle, adopt even partitioning for all operations that take place after a scan, but use Pastry-style partitioning for base data. This complexity of multiple partitionings per query has several drawbacks. First, it significantly increases the complexity of the system, and would have required extensive modification to our existing codebase. Second, it reduces the potential for exploiting data locality to increase performance; data would have to be repartitioned using the second partitioning to, say, join a table on an attribute that was partitioned by that attribute.

We decided to continue to use Pastry-style partitioning for data storage, since this preserves partitioning resiliency. However, since the data is replicated, each data item is stored at many different nodes in the system; the overlapping regions of the key space stored at multiple node are shown in Figure 5.1b. This redundancy is the key to our approach, as it means that data items are available at nodes other than the nodes that own according to the original partitioning of the key space. Furthermore, since replication is designed for seamless failover, redundant data is placed in such a way that we can adjust the boundaries of nodes' partitions in the key space by sliding them; in particular, we can slide them to reduce the load on overloaded nodes and increase the load on underloaded nodes. *This flexibility gives us significant opportunities to improve query performance by using a partitioning for query*

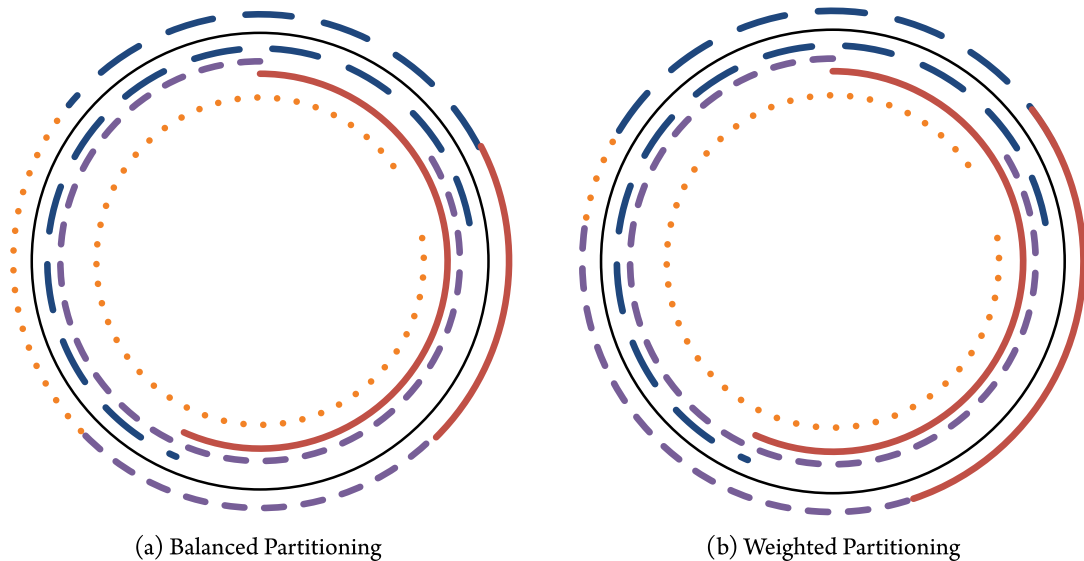


Figure 5.2: Optimized partitioning of the DHT key space. Here we show several partitionings of the key space that are possible, given the replicated data placement shown in Figure 5.1b. The overlapping colored ring segments inside the black ring show where the data is available, and the coloring of the outer ring shows which regions of the key space are assigned to which node. Figure 5.2a shows how the key space can be partitioned to assign equal amounts of data to each of the four nodes, and Figure 5.2b shows how it can be partitioned to give much less data to node n_4 (represented by the dotted orange lines). These partitionings respect the data availability due to replication.

execution that is much more balanced. We can tweak the regions of the key space assigned to each node to create a more balanced partitioning; by ensuring that the region assigned to each node falls within the region of the key space that it has *replicated* data for, we retain the ability to scan persistent storage. This eliminates the need for two different partitionings that was a problem with the above proposal to use totally even partitioning only for intermediate results. Figure 5.2a shows how the redundancy in Figure 5.1b can be used to create a more even partitioning; each each node has an exactly even fraction of the key space, giving a routing imbalance of one.

There are of course occasions where the routing imbalance is not as strongly correlated with performance. For example, if nodes are heterogeneous, then we want to allocate more of the key space to more powerful nodes, and less to less powerful nodes, in proportion to some sort of *capability* metric that is directly proportional to query execution speed at each node. Suppose that, of the nodes shown in Figure 5.1b, node n_4 is approximately six times less powerful than the other nodes, therefore has capability ⅙ of theirs. Figure 5.2b shows a partitioning that assigns approximately ⅙ as much of the

key space to node n_4 (the dotted orange regions) than to each of the other nodes. In this case, instead of creating a partitioning with a lower routing imbalance, we want to create a partitioning with a lower *weighted routing imbalance*, which considers the node with the largest fraction of the key space divided by its capability; we then normalize this by multiplying by the sum of all nodes' capabilities. This metric ranges from one (optimal) to the quotient of the sum of all nodes' capabilities and the least powerful node's capability (worst outcome). Figure 5.2b shows a partitioning that minimizes this quantity to one. Node n_4 has 1⁄19 of the key space, giving its weighted routing imbalance as $\left(\frac{1}{19} \div 1\right) \times 19 = 1$, and the other nodes have their routing imbalance as $\left(\frac{6}{19} \div 6\right) \times 19 = 1$; the maximum of these is one, so the overall routing imbalance is also one.

Another case case where naïve routing imbalance is not ideal occurs when data is skewed. Here, the implicit assumption that data points are evenly distributed through the key space is no longer true. Therefore, instead of considering fractions of the key space when computing the routing imbalance, we must consider fractions of all data. This is a very difficult problem when considering skew in the distribution of intermediate results or different skews in different relations; we want to continue to use only one partitioning during query execution, as explained above. Therefore, we consider data skew only to reduce the scanning cost of a single relation, when that cost may be the dominant component of query execution cost.

In the above examples, the data was highly replicated. Each portion of the key space was stored at ¾ of the nodes, giving us considerable leeway in choosing a partitioning. In general, however, each portion of the key space may be stored at a much smaller fraction of the nodes. There may be a string of successive nodes with IDs very close together, meaning that some nodes may not have anywhere near their "fair share" of the key space. In the more general setting, it is not clear what an algorithm to partition the key space optimally would look like. Decisions made locally (i.e. to assign a certain region of the key space to a particular node) can have global effects by forcing overly large or small regions of the key space to be assigned to other nodes because of the constraints imposed by data availability. However, the problem is relatively simply specified: we want to minimize the (weighted) routing imbalance, subject to the constraint that each node only be assigned regions of the key space it has data for. There are general software tools (and indeed an entire field of computer science research) devoted to the general problems of optimization and constraint satisfaction. In the next section, we describe how we have formulated this problem for such a tool.

5.2 Partitioning as Constrained Optimization

The general *constraint satisfaction problem* (CSP) is a well-studied problem in computer science (Tsang, 1993; Dechter, 2003; Apt, 2003). An instance of the CSP consists of a set of variables, each with its own domain, and a set of constraints; the goal is to find an assignment of all the variables to values from their domains that satisfies all of the constraints. There are many variants of this problem, which impose different conditions on the variables and constraints. A related problem is the *constrained optimization problem*, which is a CSP with an associated objective function, which should be maximized or minimized. Perhaps the best-known case of constrained optimization is linear programming (Cormen et al., 2001, chapter 29), which is widely used and studied in many computer science classes.

Partitioning the key space is an optimization problem. It is not enough to simply require that each node only be responsible for data it has a copy of; the initial Pastry partitioning will do precisely that. We want to minimize the routing imbalance, subject to those constraints. However, linear programming is not sufficiently powerful to express the definition of routing imbalance. A linear programming objective function must be a weighted sum of a collection of variables, while routing imbalance takes the *maximum* of a collection of variables. Our problem is therefore an instance of the more general constrained optimization problem, and we must look to more general software to solve it; unfortunately, with increased expressiveness typically comes increased difficulty to find a solution.

Here, we show how to formalize our partitioning problem for the Minion[1] constraint solver. Minion was chosen for a proof-of-concept implementation because it is modern, open-source, and well-documented. Gent et al. (2006) provides details of how Minion is implemented; however, the Minion interface consists almost exclusively of high-level constraint specifications, so no knowledge of how it the problem is solved is necessary to understand our formalization of the problem. While we show Minion syntax, the only requirements we make of the solver is that its input language can express the following operations in a constrained optimization problem:

- Subtract one integer from another
- Divide one integer by another
- Choose the minimum from a collection of integers

[1] Available from `http://minion.sourceforge.net`

- Enforce strict inequality of integers (i.e. $x \lneq y$)
- Enforce non-strict equality of integers (i.e. $x \leq y$)

It seems unlikely that any robust constraint solver would lack these operations. We propose to explore performance of other constraint solvers on this problem as an interesting area of future research.

Partitioning granularity

In optimization problems, there is a trade-off between the ability to find an optimal solution and the ability to find an acceptable solution quickly. A more complex formulation of a problem entails a larger search space, which takes much longer to explore fully; a simpler formulation one may never be able to find as good a solution, but the solver can find the best solution (or one close to it) *for that formulation* very quickly. In other words, a simpler model is faster but may be less representative of flexible, and therefore will produce results further from optimal.

Our optimization problem is no different. Here, the search space is the possible assignments of regions of the key space to nodes. A naïve implementation partitioning optimization would search all possible assignments using the systems native 160-bit integers as the DHT key space. This, however, leads to a needlessly large search space; the difference in routing imbalance between assigning the range `[0x00000000000000000000, 0x10000000000000000000)` to a node and assigning the range `[0x00000000000000000000, 0x10000000000000000600)` to a node is like likely to be very small. Additionally, minion (we used version 0.9) represented integers internally as signed 32-bit quantities, and therefore cannot store numbers larger than $2^{31} - 1$. Even if it supported arbitrarily large integers, performance would likely be unacceptable due to the lack of hardware support for very large numbers.

Therefore, we choose to break the DHT key space into a much smaller number of *chunks*. Chunks are assumed (for the purposes of optimization) to contain equal amount of data. If the data is approximately evenly distributed through the key space (which is very likely due to the high quality SHA-1 hash used in ORCHESTRA), then each chunk contains an equal fraction of the DHT keys space. If the data is skewed within the key space, then we can use an equi-depth histogram of the distribution of the data over the key space to split the key space into chunks. Either way, once the key space is partitioned into these chunks, the optimization problem becomes to assign these chunks to the nodes in the system.

There is still a trade-off between finding the least imbalanced partitioning possible and finding an acceptably good one quickly. Dividing the ring into a larger number of chunks leads to a larger search space, and therefore may take longer to find an acceptable solution; it may also, of course, find good solutions that a problem formulation of coarser granularity would miss. We explore the performance of different numbers of chunks experimentally in Section 5.3. For most results presented in this chapter, we choose the number of ring chunks to be five times the number of nodes; this seems to give a good trade-off between result quality and optimization speed. If the nodes have varying capabilities, as discussed in Section 5.1 and shown in the example of Figure 5.2b, then the number of chunks per node should be at least the ratio of the capabilities of the fastest and slowest nodes; however, to avoid creating an overly complex formulation of the problem, it may be desirable to cap the number of chunks per node to ensure that a reasonable solution can be found quickly.

Problem formulation

Let us begin by defining the inputs to the problem.

- The number of chunks, *numChunks*
- The counterclockwise bounds of each chunk, $chunkBottom[i]$, in the DHT key space. The clockwise bound of each chunk is implicitly the counterclockwise bound of the succeeding chunk. Each chunk therefore holds the range $[chunkBottom[(i-1) \mod numChunks], chunkBottom[i])$ of the key space.
- The number of nodes, *numNodes*
- The counterclockwise and clockwise bounds of the region of the key space available at a node n_i, $lower[i]$ and $upper[i]$, respectively
- The capability of each node n_i, $capability[i]$. Recall that is is a number directly proportional to the speed of processing at n_i. Capabilities should be scaled by a sufficiently large factor to avoid loss of precision when capability is subject to integer division by an integer in the range $[0, numChunks)$.

For the remainder of this chapter, we use $\oplus$ and $\ominus$ to denote modular arithmetic, relative to either the number of nodes or the number of chunks depending on the context.

The output of the problem is logically the partitioning of chunks to nodes. To ease problem specification, we instead use as output the *pivots*, the points around the ring that are the edges of partitions:

- *pivots*[*i*] holds the index of the most clockwise chunk that is assigned to node n_i instead of node $n_{i \oplus 1}$

As hinted at above, our optimization problem is expressed using only integers; like many other constraint solvers, Minion does not support floating-point numbers. Therefore, we cannot natively express that we want to minimize the routing imbalance. We instead chose to maximize what we term the *availability*, which now define.

Definition 10 (Node availability). *For a node* n_i*, the* node availability *is that node's capability divided by the number of chunks assigned to it. A default capability is used if the nodes are all equally capable. As mentioned above, the capabilities are made sufficiently large that there is not a serious loss of precision when the capability is divided by the number of chunks using integer division. Node availability is proportional to the inverse of the routing imbalance at anode for any particular problem instance.*

More formally, for a node n_i*,* $0 \leq i <$ *numNodes, we define*

$$\mathit{nodeAvailability}[i] \equiv \frac{\mathit{capability}[i]}{\mathit{pivots}[i \oplus 1] \ominus \mathit{pivots}[i]}$$

Definition 11 (Availability). Availability *is the minimum node availability over all nodes. This corresponds to routing imbalance for a partitioning taking the maximum of the routing imbalances at the nodes.*

$$\mathit{availability} \equiv \min_{0 \leq i < \mathit{numNodes}} \mathit{nodeAvailability}[i]$$

The objective function of our optimization function is, therefore, to maximize the availability of the partitioning. We now turn to the constraints. We first describe how we translate the available regions of each node specified by *lower* and *upper* into chunk numbers by rounding in the appropriate directions. For $0 \leq i <$ *numNodes*,

$$\begin{aligned} \mathit{lowerChunk}[i] &\equiv \text{index in } \mathit{chunkBottom} \text{ of first ID clockwise from } \mathit{lower}[i] \\ \mathit{upperChunk}[i] &\equiv (\text{index in } \mathit{chunkBottom} \text{ of first ID clockwise from } \mathit{upper}[i]) \ominus 1 \end{aligned}$$

Note that this definition is conservative and correct, in that it ignores partial chunks. As the number of chunks is reduced, the fraction of each node's available range that is ignored because it is in partial chunks increases; this is why some accuracy is sacrificed when the number of chunks is reduced.

After this "chunkification" is complete, we can express the availability constraints (to enforce that each node can only be assigned portions of the key space for which it has a copy of the data). We would like to simply state that each pivot (where the key space switches from one node to the next) be in the region of overlap between those two nodes, i.e.

$$\mathit{pivots}[i] \in [\mathit{lowerChunk}[i], \mathit{upperChunk}[i]] \cap [\mathit{lowerChunk}[i \oplus 1], \mathit{upperChunk}[i \oplus 1]]$$

Unfortunately, there is an added layer of complexity: the overlap range may wrap around zero, and disjunction is not expressible (i.e. one cannot state that the pivot must be greater than 12 or less than 3). There can be at most one node whose owned region in the original Pastry partitioning wrapped around zero. If we assume that the replication factor is less than the number of nodes (otherwise all nodes have all data and this is not a very constrained optimization problem!), then there must be a node whose available region does *not* include the region owned by the wrapping node. Therefore there is at least one node whose available region does not wrap around zero. Moving counterclockwise from this node to other nodes, eventually their available ranges may start to wrap around zero; the same hold moving clockwise. Therefore, if we number the nodes in increasing order by ID (with n_0 as the first clockwise from zero), we can partition the nodes as follows

$$\begin{aligned}
\mathit{wrapsDown} &\equiv \{i \| \mathit{lowerChunk}[i], \mathit{upperChunk}[i]) \text{ wraps around } 0 \wedge \forall\, j \in \mathit{noWrap}\ i < j\} \\
\mathit{noWrap} &\equiv \{i \| [\mathit{lowerChunk}[i], \mathit{upperChunk}[i]) \text{ does not wrap around } 0\} \\
\mathit{wrapsUp} &\equiv \{i \| \mathit{lowerChunk}[i], \mathit{upperChunk}[i]) \text{ wraps around } 0 \wedge \forall\, j \in \mathit{noWrap}\ i > j\}
\end{aligned}$$

and can guarantee that $\mathit{noWrap} \neq \emptyset$; *wrapsDown* and *wrapsUp* may be empty.

For the purposes of expressing overlapping regions, we can therefore "unroll" the key space ring by going around it three times. We express the overlap between nodes in *noWrap* in the "primary" circuit around the ring; overlap from *wrapsDown* that wraps begins before the primary circuit extends into the "down" circuit from the primary circuit, and overlap from *wrapsUp* that finishes after the primary circuit extends into the "up" circuit. The three circuits are numbered sequentially, beginning

with the down circuit, using the supplied division of key space into chunks. We then require that each pivot be in the overlapping regions, as expressed over those three circuits.

Example 5. Suppose we have the four nodes shown in 5.1b, and the DHT ring is broken into 20 chunks. For the purposes of this example we number the nodes from 1, as they are in the figure, rather than from 0. Then $\textit{wrapsDown} = \{1\}$, $\textit{noWrap} = \{2, 3\}$, and $\textit{wrapsUp} = \{4\}$. Since we go around the ring three times, pivot values can range from 0 to 60. $\textit{pivots}[1]$, where assignment switches from n_1 to n_2, can take place anywhere in their overlap, $[0, 4)$. Since 2 is in *noWrap*, we require that $20 \leq \textit{pivots}[0] < 24$. Continuing around the ring, we generate the following complete set of constraints. In our implementation, for simplicity's sake we store an extra pivot, which is required to be one complete circuit around the ring from the first pivot; this is also shown here. Note that allowable values of `pivots`[4] do wrap around zero.

$$\begin{array}{rcccl} 20 & \leq & \textit{pivots}[1] & < & 24 \\ 24 & \leq & \textit{pivots}[2] & < & 30 \\ 25 & \leq & \textit{pivots}[3] & < & 40 \\ 32 & \leq & \textit{pivots}[4] & < & 42 \\ 40 & \leq & \textit{pivots}[5] & < & 44 \end{array}$$

From here, the only additional constraint needed is that the pivots progress around our unrolled ring, i.e. $\textit{pivots}[i] < \textit{pivots}[i+1]$. The size of the generated problem is linear in the number of nodes, not surprisingly, since all of the constraints are on the pivots between nodes. Now that we have seen what how optimized partitioning is expressed as constrained optimization in general, let us examine a concrete example.

Constrained Optimization in Minion

Figure 5.3 shows a Minion input file generated by our implementation for the scenario in the previous example (shown in Figure 5.1b). The `VARIABLES` section defines all of the variables that will be used in the search *and* in the computation of the objective function; the objective function is also calculated using constraints. We use the default capability value of 1,000,000 for all nodes. All of the arrays are indexed from zero. The `size` array holds the number of chunks assigned to each nodes. The `SEARCH` section simply says to search through possible assignments to the `pivots` array, and

```
MINION 3

**VARIABLES**
BOUND pivots[5] {0..60}
BOUND nodeavailability[4] {0..1000000}
BOUND size[4] {0..20}
BOUND availability {0..1000000}

**SEARCH**
VARORDER [pivots]
PRINT [[pivots]]
MAXIMIZING availability

**CONSTRAINTS**
min(nodeavailability, availability)
difference(pivots[4],pivots[0],20)
ineq(pivots[0],pivots[1], -1)
ineq(pivots[1],pivots[2], -1)
ineq(pivots[2],pivots[3], -1)
ineq(pivots[3],pivots[4], -1)
difference(pivots[1],pivots[0],size[0])
difference(pivots[2],pivots[1],size[1])
difference(pivots[3],pivots[2],size[2])
difference(pivots[4],pivots[3],size[3])
div(1000000,size[0],nodeavailability[0])
div(1000000,size[1],nodeavailability[1])
div(1000000,size[2],nodeavailability[2])
div(1000000,size[3],nodeavailability[3])
ineq(1,pivots[0], 0)
ineq(pivots[0],3,-1)
ineq(4,pivots[1], 0)
ineq(pivots[1],10,-1)
ineq(5,pivots[2], 0)
ineq(pivots[2],19,-1)
ineq(12,pivots[3], 0)
ineq(pivots[3],22,-1)
ineq(21,pivots[4], 0)
ineq(pivots[4],23,-1)

**EOF**
```

Figure 5.3: Example Minion input file. This balancing program attempts to distribute the DHT key space as evenly as possible among the four nodes, given the replicated data placement shown in Figure 5.1b.

to print out the results, while maximizing the `availability` variable; availability is as defined above. The interesting part of the input file is in the `CONSTRAINTS` section, unsurprisingly. The `min` line defines the overall availability used in the function as the minimum over all individual node availabilities. The `difference` line enforces that the first and last pivots be one full cycle around the ring (i.e. 20 chunks) apart. The following `ineq` lines enforce that the pivots be in ascending order. The `difference` lines enforce that the size array holds the number of chunks assigned to each node. The `div` lines ensure that the nodeavailability array holds the availability of each node; if the nodes were not equally capable, their capabilities, included here, would not all be the same. The final set of `ineq` lines ensures that the pivots are in allowable locations, as specified above.

The restrictions on the pivots are slightly different that given above, for two reasons. One is that our constraint generator will start the primary circuit at zero if nothing falls into the down circuit, as in this case. The second is that it naïvely assumes that the edges of intersection regions do not fall *exactly* on chunk boundaries, and so always rounds conservatively. In this case, due to the synthetic nature of the node IDs, they do, but this is extremely unlikely with the pseudorandom node IDs that result from hashing.

Limitations

This approach has several limitations. One is that each node will be assigned at least one chunk, even if this increases the weighted node imbalance; this is necessary because otherwise the division operation will fail. Very underpowered nodes could be removed in a preprocessing step. In a related problem, nodes that have available less than one chunk of data will cause the constraints to be insoluble; this would require that, say $7 \leq \mathit{pivots}[\mathrm{i}] < 7$. These will also have to be removed in a preprocessing step. It would be desirable if both of these problems could be addressed *inside* the generated problem instead of as a modification to its input. Neither problem arose while we were performing our experimental analysis of performance, and so we did not pursue these issues further.

A second limitation of this approach is that, if the optimizer cannot find an optimal solution, either due to time constraints (likely due to the very large search space), or due to inherent unevenness of the data distribution among nodes, the objective function does not try to ensure evenness among the remaining nodes. This is desirable to reduce the effect of slowdown causes by transient load.

Consider the following example. Suppose we have a system with 20 equally capable nodes. Half

have a per-node routing imbalance of 1.2, and half 0.8, meaning that the half of them are underloaded. The overall partitioning has a routing imbalance of 1.2. Now consider a second system, also of 20 equally capable nodes, where one has per-node routing imbalance 1.2, one 0.8, and the remaining 18 have per-node routing imbalance 1.0. The overall partitioning still has routing imbalance 1.2. Suppose that with perfect partitioning, a query would execute in 10 seconds. With either of these partitions, we expect the query to take 12 seconds to execute, since some nodes are performing 20% more work than they would in perfect partitioning.

Suppose, then, a single randomly chosen node's execution is slowed (due to "last-mile" network congestion, or transient load at that node) by a factor of 10%. In the first instance, with probability ½, an already-overloaded node will be slowed, causing that node to take $10 * 1.2 * 1.1 = 13.2$ seconds to execute the query, and therefore being the bottleneck. With probability ½, an underloaded node will be slowed, and query completion time is not slowed. In the second instance, only slowing the overloaded node will delay query completion, with probability 1/20. In general, it is desirable to minimize the number of overloaded nodes.

For future work, we propose to explore approaches to solve both of these limitations. In particular, however, it is not immediately obvious how to alter the constraint solver's objective function. We discuss this and other ways to improve the techniques presented here in Section 7.3.

5.3 Experimental Analysis

We conducted an extensive experimental evaluation of several aspects of load balancing. We wanted to explore the costs and benefits of balancing using constrained optimization on routing imbalance, to verify if improvements were in fact generally possible. We also wanted to explore the effects of various parameters, such as execution time, replication factor, and partitioning granularity on result quality. These results are presented in Section 5.3. We also wanted to verify that optimized partitioning improved query performance in the various cases outlined earlier. Section 5.3 begins with a comparison of query execution using partitioning generated by the optimization techniques described previously with more traditional Pastry-style partitioning and the totally even partitioning from Chapter 4. This is followed by the results of a series of experiments on query performance when executing over heterogeneous nodes. We then discuss the effect of taking skew into account when creating optimized partitionings. We conclude our experimental analysis with a comparison of Orchestra query ex-

ecution using partitioning optimization with several more-traditional RDBMSs, now that we have solved the major shortcoming of Chapter 4. We compare both with large databases on high-powered servers, and with proportionally smaller databases running on a lower-powered node, to give an idea of the overhead imposed by our query processing and data storage layers.

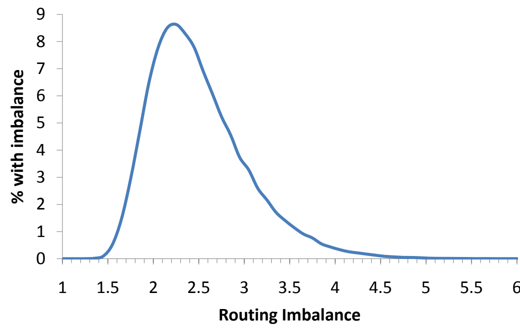
Balancing Quality

We begin our experimental analysis with a verification that the Minion optimizer can significantly decrease the routing imbalance from the baseline (standard Pastry partitioning) in a reasonable period of time. These experiments were all performed on an 2.83 GHz Intel Core2 Quad with 4GB RAM running Windows Vista. It should be noted, however, that Minion is single-threaded and typically used about 32MB RAM; we expect similar performance results on less capable machines of a similar clock speed.

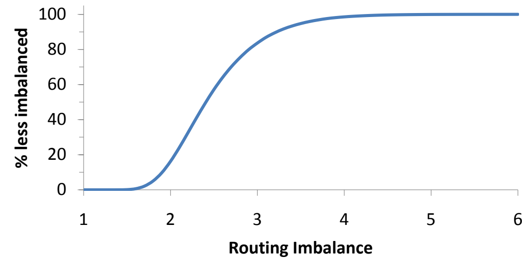
Imbalance of Initial Partitionings

We set out to quantify the amount of imbalance present in basic Pastry-style partitionings for the system sizes we're interested in. Clearly, if most partitionings are approximately balanced already, then any attempt to create better partitionings would be wasted effort. We show histograms of routing imbalance from an experiment using random node IDs (as would likely occur in our system, where we take the SHA-1 hash of a node's IP address and port) on the left side of Figure 5.4 for 20, 40, and 100 nodes. Most partitionings are somewhat even; no single node is responsible for, say, half the data, which would require a routing imbalance of 10 in Figure 5.4a and 50 in Figure 5.4e. However, it is clear that for smaller systems imbalance is often at least 2 (implying queries take approximately twice as long as they would in truly even partitioning), and for larger systems imbalance is often at least 3. The right side of Figure 5.4 gives cumulative histograms for the same data, to show what fraction of the Pastry-style partitionings for each system size are more even that each routing imbalance. Finally, Figure 5.5 shows summary statistics of the distribution of routing imbalances as the number of nodes increases.

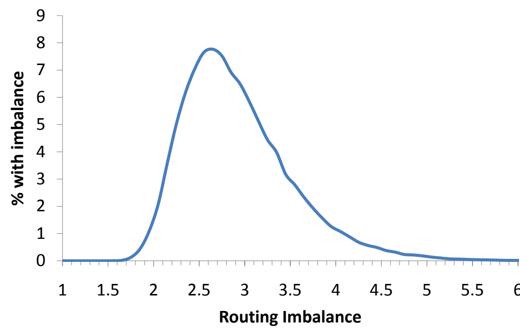
In general, it is clear that initial Pastry-style partitionings slowly become more imbalanced as the size of system increases. This is not surprising, as the more (randomly chosen) node IDs are used to partition the key space, the more likely it is that two adjacent ones will be (relatively) very far



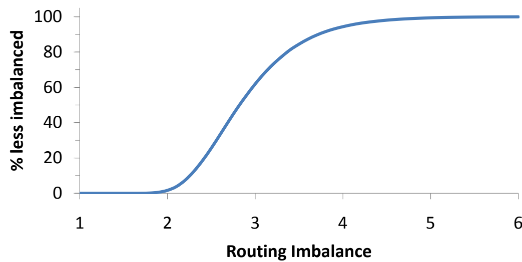
(a) Partitioning quality histogram, 20 nodes



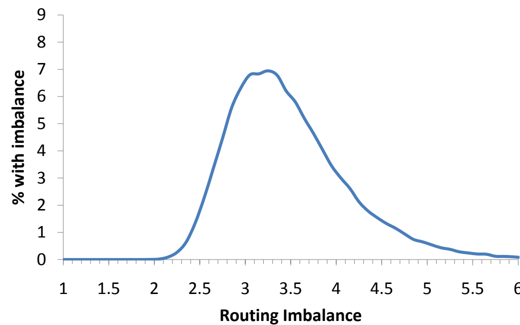
(b) Partitioning quality cumulative histogram, 20 nodes



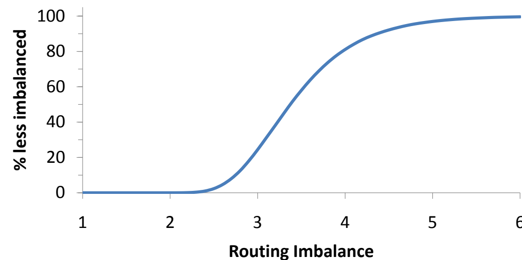
(c) Partitioning quality histogram, 40 nodes



(d) Partitioning quality cumulative histogram, 40 nodes



(e) Partitioning quality histogram, 100 nodes



(f) Partitioning quality cumulative histogram, 100 nodes

Figure 5.4: Initial quality of partitionings for 20, 40, and 100 nodes with randomly generated node IDs. For the default Pastry-style partitionings, we show the percentage of all partitionings created that have or have at most the specified imbalance. Lower imbalance implies a more even, higher quality partitioning. We show results from 100,000 trials for each system size. Bucket size for the histograms is 0.1.

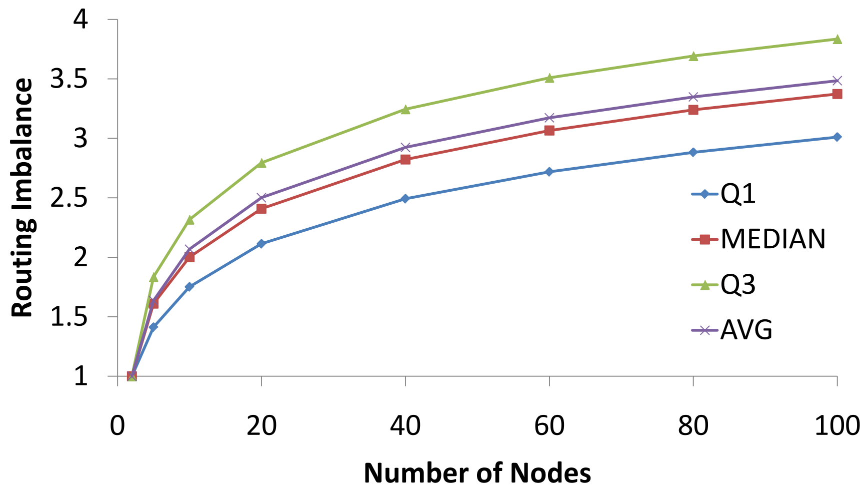
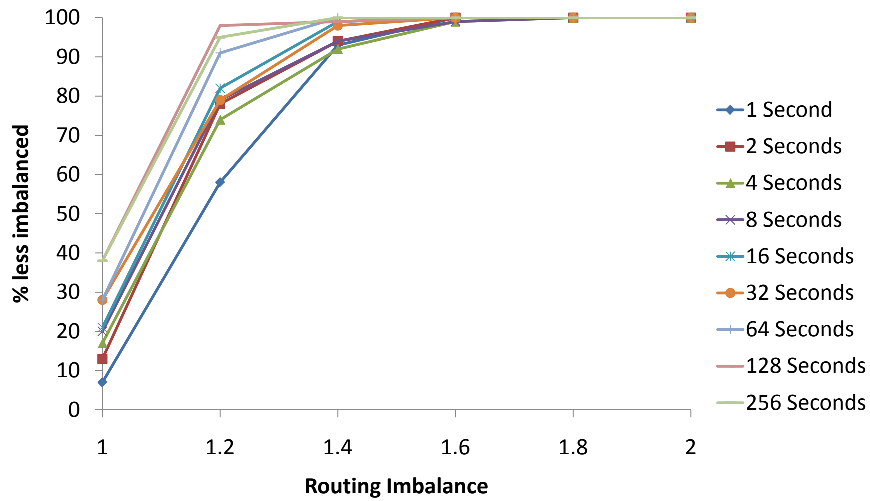


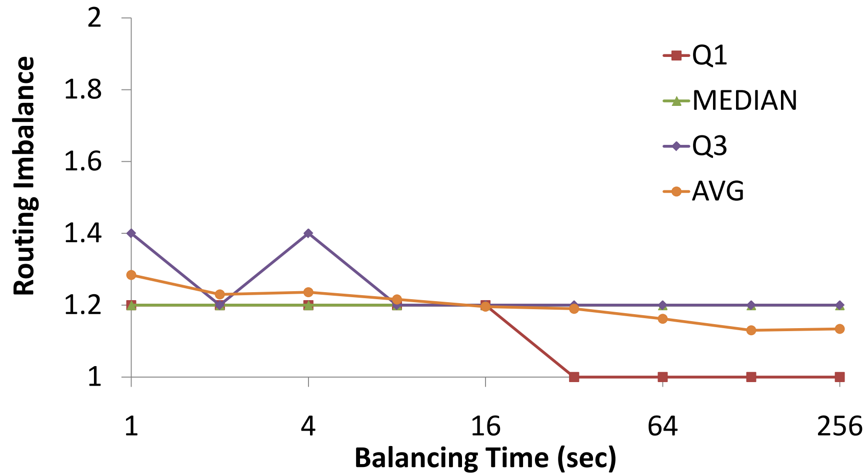
Figure 5.5: Initial quality of partitioning as system size changes. We plot summary statistics (first and third quartiles, median, and mean) as the number of nodes goes from 2 (where partitioning is inherently even) to 100. Note that while the amount of imbalance increases along with the number of nodes, adding nodes to a system will never increase the amount of data at a given node. We show results from 100,000 trials for each system size.

apart, increasing the imbalance relative to optimal. It is important to remember, however, that adding nodes while keeping the data size fixed will never increase the amount of data stored at any node (or therefore the most loaded node); it merely increases the imbalance relative to ideal, totally even data partitioning. If the size of the system is very large relative to the amount of data, then data imbalance is unlikely to be a cause of poor query performance. However, if one were to increase the amount of data in the system in direct proportion to the number of nodes, one would expect execution time of a query to increase; the increased routing imbalance will mean that the amount of data stored at the most loaded node will increase, making it more of a bottleneck in query execution.

We observe that for all system sizes greater than or equal to ten, the average (and median) partitioning has imbalance greater than two. Therefore, there is significant room for improvement even in modestly sized systems. In larger systems, there is the potential for even larger gains relative to the baseline Pastry partitioning; at 100 nodes ¾ of the partitionings had imbalance more than three. Given these potential benefits in query execution speed, even relatively expensive balancing operations can be justified, especially if amortized of a number of query executions.



(a) Cumulative histogram



(b) Summary statistics

Figure 5.6: For optimization times between 1 and 256 seconds, we show the effect of optimization time on balancing quality. Lower imbalance implies a more even, higher quality partitioning. We performed 100 trials for each balancing time, with 20 nodes with randomly generated IDs and replication factor 5. We show a cumulative histogram of the percentage of partitionings created that are at least as good as each imbalance amount, and summary statistics (first and third quartiles, median, and mean) of the imbalances for each amount of optimization time.

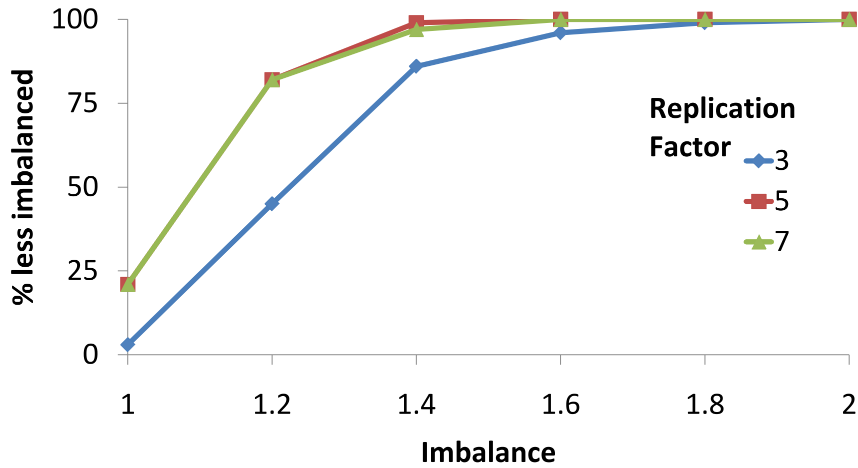
Imbalance and Optimization Time

Next, we set out to study how long it takes to created a reasonably good partitioning of the key space, and how much the amount of time given to the constraint solver to work on the problem affects the quality of the output. We used 20 randomly generated node IDs, and assumed data was stored with replication factor 5. We use 5 partitioning chunks for each node, meaning that the key space was divided into 100 total chunks. Routing imbalance is, in this context, always a value of the form $\frac{i}{5}$, where the integer $i \geq 5$; routing imbalance 1.2 means that at least one node was assigned more than its even share of chunks. We vary the time budget given to the optimizer from one second to 256 seconds. Figure 5.6a shows cumulative histogram for routing imbalance for various optimization times, and Figure 5.6b show summary statistics for routing imbalance as a function of time.

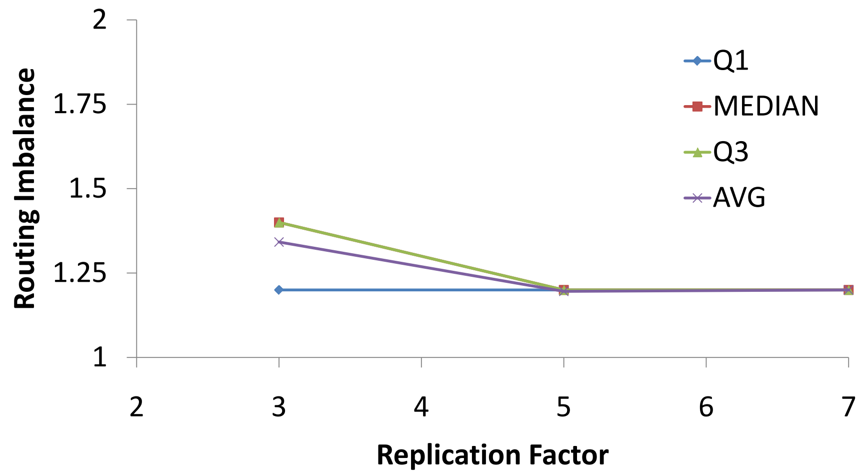
From these results, we can draw several conclusions. Even for virtually no optimization time, the average routing imbalance was less than 1.3, as shown in Figure 5.6b; in contrast, as shown in Figure 5.5, the average routing imbalance for the Pastry baseline Pastry with 20 nodes is approximately 2.2, and 75% of partitionings have imbalance greater than 2. Therefore partitioning has been greatly improved. If we are willing to devote even slightly more time to optimization, then routing imbalance can be reduced even further. From Figure 5.6a, we see that for the still modest optimization times of 2 to 16 seconds, it is possible to achieve routing imbalance of at most 1.2 70-80% of time. Imbalance was worst than 1.4 at most 10% of the time, and was virtually never worse than 1.6. All of these represent a significant improvement from the baseline. These results verify that significant decreases in routing imbalance are possible, and that much of the benefit comes relatively at a relatively small cost.

Imbalance and Replication Factor

We were also interested in how data replication affects our approach's ability to balance the partitioning. Figure 5.7 shows a comparison of replication factors 3, 5, and 7 for 20 nodes and relatively limited optimization time (16 seconds). Recall that, in our system, the replication factor r must be odd; data is replicated at $\lfloor \frac{r}{2} \rfloor$ nodes on each side of the node that owns the data in the original partitioning. In this scenario, we see that replication factor 7 offers no benefit beyond that afforded by replication factor 5. Our approach does not produce as good partitionings for replication factor 3; the significantly reduced amount of replication constrains the system to the point where better balancing is not

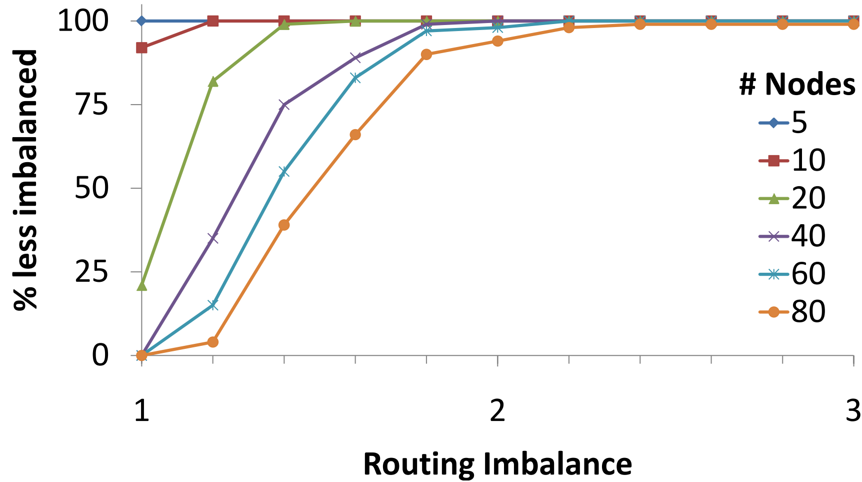


(a) Cumulative histogram

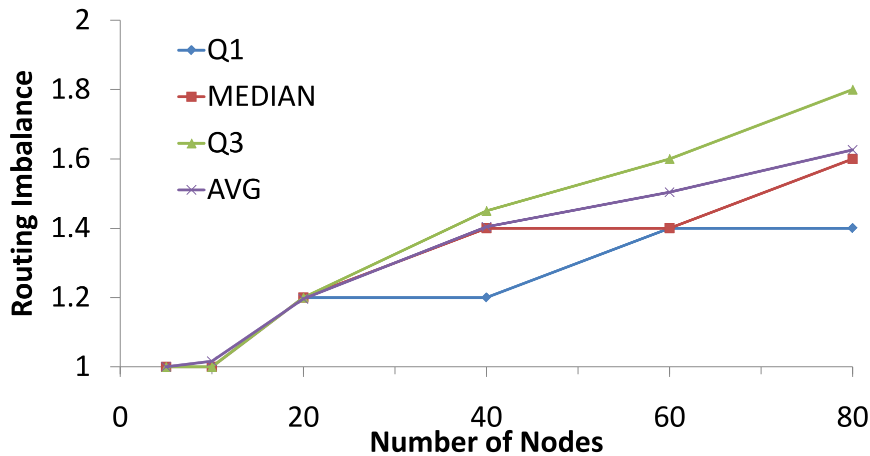


(b) Summary statistics

Figure 5.7: Effect of replication factor on balancing quality. Lower imbalance implies a more even, higher quality partitioning. We performed 100 trials each for replication factors 3, 5, and 7, with each using 20 nodes with randomly generated IDs and 16 seconds optimization time. We show a cumulative histogram and summary statistics (quartiles, median, and mean) of the imbalances for each replication factor. Recall that, since we use five chunks per node, imbalance 1.2 means that at least one node was assigned an extra chunk.



(a) Cumulative histogram



(b) Summary statistics

Figure 5.8: Effect of number of nodes on balancing quality. Lower imbalance implies a more even, higher quality partitioning. We performed 100 trials each for 20 to 100 nodes with randomly generated IDs and 16 seconds optimization time. We show a cumulative histogram and summary statistics (quartiles, median, and mean) of the imbalances for each system size.

possible. We see, however, that even the limited balancing possible can be a significant improvement from the baseline, where imbalance for 20 nodes was often at least 2 (as shown in Figure 5.5).

Optimization for Larger Systems

The balancing experiments presented so far have focused on systems with 20 nodes. This was chosen because we feel it is a reasonable size of an Orchestra instance. However, we would like to explore what happens to the quality of optimized partitionings as the system size increases. As we saw before

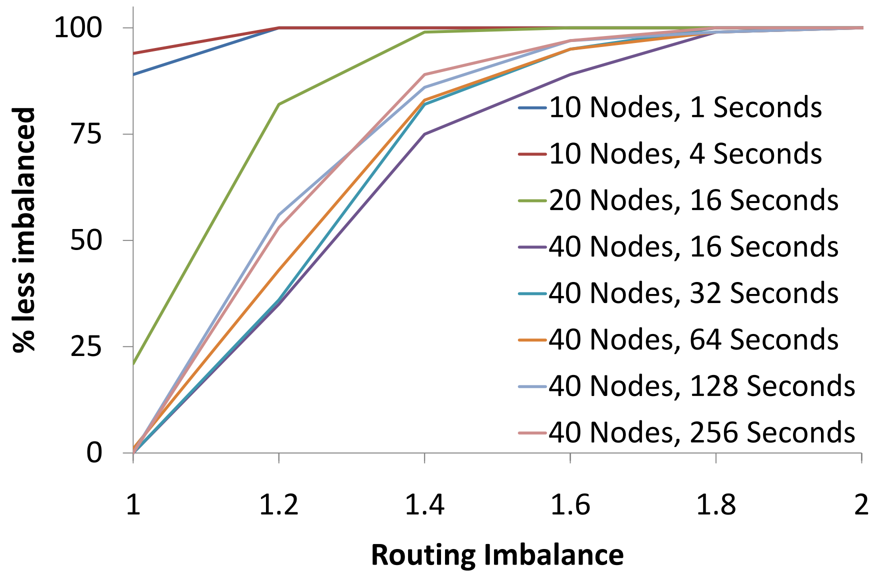


Figure 5.9: Effect of number of nodes on optimization time. Increasing the number of nodes requires both an increase in optimization time and leads to inherently greater imbalances.

in Figure 5.5, larger systems have greater imbalance in the baseline Pastry partitionings. Figure 5.8 shows the results of this experiment. As we vary the system size (while holding the replication factor at 5), we see that larger system sizes result in a less evenly balanced partitioning. This is not surprising, given that we hold the replication factor constant; therefore larger systems have proportionally less partitioning flexibility, as each data item is stored at a smaller fraction of the nodes. With five nodes, each data item is stored at every node, giving the system complete partitioning flexibility, but with 80 nodes, each data item is stored only at 1⁄16 of them. Additionally, the more nodes there are, the more likely that a number of them will be very close together; this will cause some of the n nodes to have less than $\frac{1}{n}$ of the data, making even partitioning impossible.

Given that increasing the number of nodes also significantly increases the search space (since we hold the number of chunks per node constant at 5), we wanted to verify that larger systems were inherently more imbalanced, instead of just more difficult to balance. We therefore performed another series of experiments to vary the number of nodes and the amount of time given to the constraint solver. The results of this experiment are shown in Figure 5.9 as cumulative histograms of routing imbalance for a variety of optimization times and system sizes, including our baseline of 20 nodes allotted 16 seconds. While increasing the amount of execution time does improve performance for larger systems, even when execution time is increased by a factor of 16 we see that imbalance for

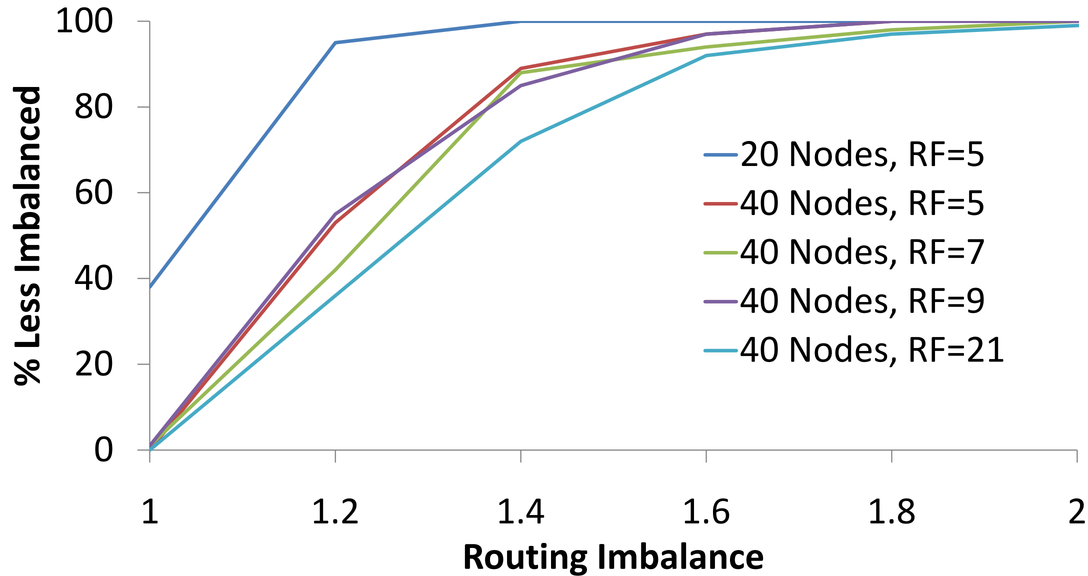


Figure 5.10: Effect of changing replication factor for 40 nodes. We kept optimization time at 256 seconds, and continued to use 5 chunks for each node. We see that increasing replication for 40 nodes never creates balancing as good as for 20 nodes, even when each data item is stored at approximately half of all nodes. We conclude that the constraint solver is not performing well given a larger search space.

40 nodes is never as low as it is for 20 nodes; similarly, 10 nodes can be optimized much better than 20 nodes, even when given 1⁄16 as much time.

Given our concerns about constraint solver performance, and to better understand the amount of replication needed in larger systems to ensure that we can create partitionings as even as those for 20 nods, we decided to also explore more complex scenarios. Figure 5.10 compares baseline performance for 20 nodes at replication factor 5 with performance for 40 nodes at various replication factors. We see that increasing the replication factor for larger systems does not correspond to an increase in the quality of the resulting partitioning, and conclude that the constraint solver is not doing a good job at exploring the much larger search space. Reducing the number of chunks per node improved performance somewhat, but as this is clearly not feasible for systems much beyond 40 nodes we did not perform an exhaustive study of this and do not present such results here. It therefore may be necessary to alter the objective function or find other ways to assist (or replace) the constraint solver; we mention ways this might be accomplished in Section 7.3.

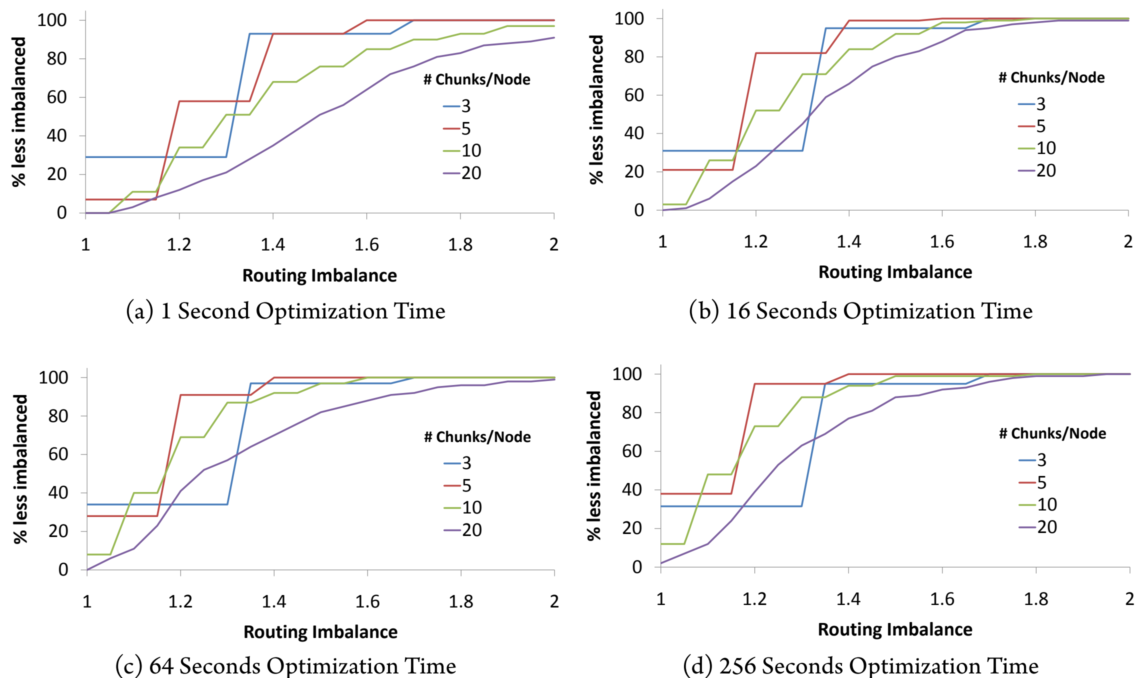


Figure 5.11: Effect of number of partitioning chunks on balancing quality. Lower imbalance implies a more even, higher quality partitioning. We performed 100 trials each for 3 to 10 chunks per nodes, with 20 nodes with randomly generated IDs. Since increasing the number of chunks increases the problem complexity, we also considered increasing the optimization time from 16 seconds to 64 or 256 seconds, a four- or eight-fold increase, respectively. We also give results when optimization time is extremely limited, at 1 second. We show a cumulative histogram for each such experiment; it is also interesting to compare results between experiments.

Necessary Number of Chunks

We have just seen that increasing the number of nodes lead to increased routing imbalance, due to both the increased problem complexity and the reduced availability of each piece of the key space. We also wanted to explore the effect of increasing problem complexity. We expect that coarse-grained partitioning of the key space will give better solutions for short optimization times, since the problem is simpler; finer-grained partitioning may allow better solutions for certain instances, since it has greater flexibility, though at the cost of a much larger solution space. We see in Figure 5.11 that this is somewhat the case. For shorter optimization times, 3 chunks per node (60 chunks total) creates the highest fraction of optimized partitionings that are totally even. However, for most instances, 5 chunks per node (100 chunks total) creates the best partitionings. Clearly the large increase in the search space does not create that many better solutions, and in general just adds complexity. We

conclude that a conservative number of chunks per node offers the best performance. Based on the results of this experiment, we used 5 chunks per node for our later experimental evaluation of query performance.

Query Execution Performance

We have shown that optimized partitioning can produce significantly more balanced partitionings than traditional Pastry partitioning. Now, we would like to explore the effect this has on query performance. Since we need to generate a new optimized partitioning only when the set of participant nodes changes, we can potentially use a partitioning for many different queries; were therefore do not consider the optimization costs here. We used the system described in Section 4.5, with the following minor modification. In our implementation, the node initiating a query already distributed a partitioning snapshot (or equivalently, a routing table with complete knowledge of all the node) along with the query plan; we simply added the ability to distribute a precomputed optimized partitioning instead of the default one. In this way, we gain the ability to use optimized partitioning without introducing any consistency issues: as before, the set of nodes known to the query initiator when the query begins participate in the query. In fact, we had to make no modifications whatsoever to the query execution implementation. While we assume here that the node that poses the query is the node that performs the partitioning optimization, in practice the nodes would likely share optimized partitionings with each other and choose to use the best one they're aware of, assuming it's up-to-date.

We compare query performance of optimized partitioning against two baselines. One is naïve Pastry partitioning, which is used in many distributed hash table implementations. It is worth remembering, of course, that practical Pastry-based implementations typically achieve load balancing (with high probability) using virtual nodes; as explained previously, this virtual node approach is not an option for us due to the index page fragmentation it creates. Pastry-based approaches do have the partitioning resiliency property described in the introduction. The totally even partitioning from Chapter 4 lacks this property. Since our optimization approach relies on (replicated) Pastry-style data layout, it is resilient; recall from the introduction to this chapter that this was a major motivation of this work. We also benchmark against totally even partitioning, despite its limitations, as it may be reasonable in low-churn settings, and it offers an upper bound on the performance that we

can hope to achieve. We expect performance of our approach to fall somewhere in between Pastry-style partitioning and totally even partitioning, and hopefully much closer to even partitioning. This means that it will combine the good performance (due to excellent load balancing) of even partitioning with the resistance to churn of Pastry-style partitioning.

We explore query performance over three sets of queries, which operate over two data sets. One data set is the standard TPC-H benchmark data, which we also used in Chapter 4. It is a well-studied benchmark in the query processing community. We use it at scale factor 10; at this size, it contains approximately 10GB data in total. Additionally, to study data skew, we use a modified TPC-D[2] data generator, developed by Surajit Chaudhuri and Vivek Narasayya of Microsoft Research (Chaudhuri and Narasayya, 1999). It alters the standard TPC-D data generator to choose all values using a Zipfian distribution, where some values are much more likely that others. We use it to explore query performance over skewed data, which as mentioned earlier is also a goal of this work. Also as in Chapter 4, we use TPC-H queries Q1, Q3, Q5, Q6, and Q10; these are single-block SQL queries that our optimizer can support.

However, these queries from the benchmark suite are all aggregate queries. While we expect that aggregate queries will show up frequently in a distributed implementation of Orchestra (such as in the computation of transaction priorities), we also expect non-aggregate queries with many join to appear; many mapping queries have this form. Specifically, mapping queries are often over a so-called star schema, where there is a main central table that all other tables join with. The TPC-H schema comes very close to this, so we use it as the basis for several star-schema queries. The first, which we call "Star Schema Query 1," and label in some of the figures as QA to differentiate it from TPC-H Q1 above, creates the "open machinery orders" relation for March 15, 1995:

```
SELECT c_name, p_name, l_quantity, o_orderdate, o_shippriority,
      l_shipdate
FROM customer, orders, lineitem, part
WHERE c_mktsegment = 'MACHINERY' AND c_custkey = o_custkey AND
      l_orderkey = o_orderkey AND p_partkey = l_partkey AND
      o_orderdate < date '1995-03-15' AND
      l_shipdate > date '1995-03-15'
```

The second, "Star Schema Query 2" or QB, creates the "international large part order" relation for the first half of 1993:

```
SELECT l_orderkey, c_name, s_name
```

[2]TPC-D and TPC-H share the same schemas and data.

```
DB1(pName : CHAR(8), speciesId : INT, desc : VARCHAR(75))
SPECIES(speciesId : int, genus : VARCHAR(20),
      species : VARCHAR(20))
```

(a) Base relations

```
DB1toDB2(pName : CHAR(8), db2Id : INT)
DB2toDB3(db2Id : int, db3Id : INT)
DB3toDB4(db3Id : int, db4Id : INT)
```

(b) Correspondence tables

```
DB4(db4Id : INT, genusSpecies : VARCHAR(41),
      desc : VARCHAR(75))
```

(c) Result relation

Figure 5.12: Relations for BioJoin benchmark

```
FROM lineitem, customer, supplier, part, orders
WHERE p_partkey = l_partkey AND c_custkey = o_custkey AND
      s_suppkey = l_suppkey AND p_partkey = l_partkey AND
      l_orderkey = o_orderkey AND p_size > 40 AND
      c_nationkey <> s_nationkey AND
      l_shipdate >= date '1993-01-01' AND
      l_shipdate < date '1993-06-01'
```

We feel these queries represent the types of queries liable to be posed over star schemas, with a variety of selective predicates being posed over the "leaf" relations, and the selection of a variety of attributes from both the central relation and the leaf relations.

However, many mappings, especially in bioinformatics settings, are expressed as correspondence tables that equate IDs in different databases. We explored a complex "chain" join that arises from the composition of several such correspondence tables (for hypothetical databases of proteins and their functions), and also performs some simple data transformation. While the databases are hypothetical, the types of transformations and data that are used in this scenario come from our experience with data integration for bioinformatics. The base data, correspondences, and, result schema are shown in Figures 5.12a, 5.12b, and 5.12c, respectively. The transformations substitute the string ID from DB1 with integer IDs for DB2, DB3, and DB4. Additionally, the separate genus and species attributes from DB1 must be combined into one attribute in DB4 using string concatenation. The query that we use to perform this is as follows:

```
SELECT db4Id, genus || ' ' || species, desc
FROM DB1 NATURAL JOIN SPECIES NATURAL JOIN DB1toDB2
```

```
NATURAL JOIN DB2toDB3 NATURAL JOIN DB3toDB4
```

For our experiments, we used 20,000 species, which were assigned uniformly at random to the proteins in DB1. We vary the sizes of the other relations together, and hold each join selectivity constant at 0.75, meaning that ¾ of the proteins in DB1 correspond to an entry in DB2, and the same for the other correspondences. We call this scenario the *BioJoin* benchmark for obvious reasons. We consider instances where each of DB1, DB1toDB2, DB2toDB3, and DB3toDB4 has size 1,000,000, 1,500,000, 2,000,000, and 2,500,000; given the selectivities, these produce DB4 instances with approximately 421,875, 632,813, 843,750, and 1,054,688 tuples, respectively.

As with other measurements of system performance in this thesis, each data point represents the average of at least 5 trials. We show 95% confidence intervals for all data points. For such confidence intervals, the standard formula gives the uncertainty of a mean $\overline{x}$ is $1.96\frac{\sigma}{\sqrt{n}}$ as a function of the sample standard deviation σ and the number of trials n, giving a range of $\overline{x} \pm 1.96\frac{\sigma}{\sqrt{n}}$.

Balanced Partitioning Query Performance

We begin with a comparison of optimized partitioning with both baseline Pastry partitioning (which has the partitioning resiliency property important for replication, but which does not distribute data that evenly) with the totally even partitioning used in Chapter 4, which distributes the data very evenly but which lacks partitioning resiliency, increasing the data movement caused by churn. As we are trying to balance the partitioning as much as possible, given data placement, we call this *balanced partitioning*. Figure 5.13 shows performance results for the TPC-H queries and the star schema queries, running on 20 nodes from Amazon's EC2 cloud computing service. We use the "large" nodes, which, despite their name, are in fact the second-smallest nodes in terms of processing power and memory: they each have 7.5GB RAM and a virtualized dual-core 2GHz Opteron CPU. As expected, the optimized balanced partitioning has performance between Pastry partitioning and our original partitioning scheme. Fortunately, it has performance only 10-15% slower than even partitioning, and much faster than Pastry-style partitioning. This validates the approach, showing that balancing is important for good performance; it also shows that the overhead of partitioning resilience is not too high. This result is approximately what we would expect, given that Pastry partitioning for 20 nodes averages a routing imbalance of about 2.5 (as in Figure 5.5), while balanced partitioning for a reasonable partitioning time (we used 16 seconds) for 20 nodes averages about 1.2, as shown

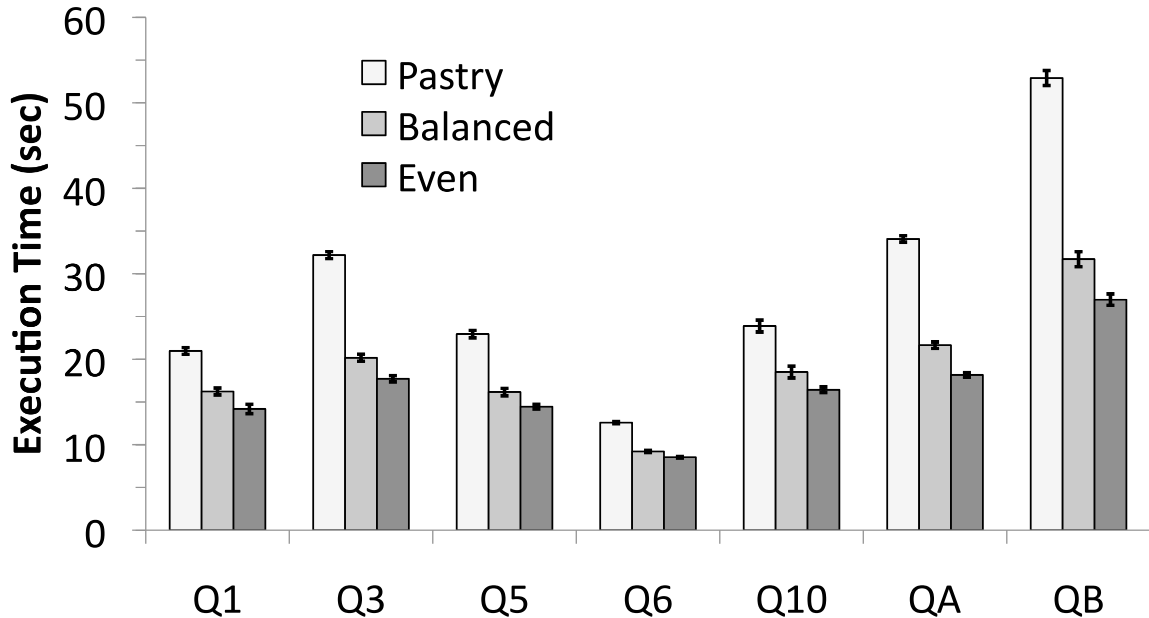
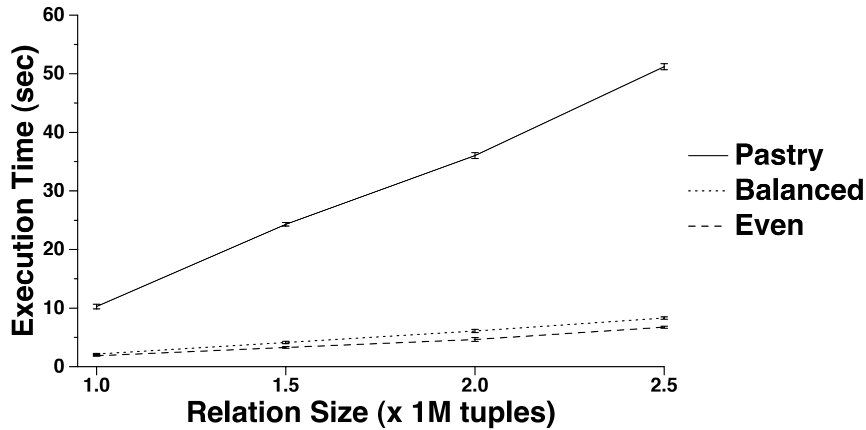


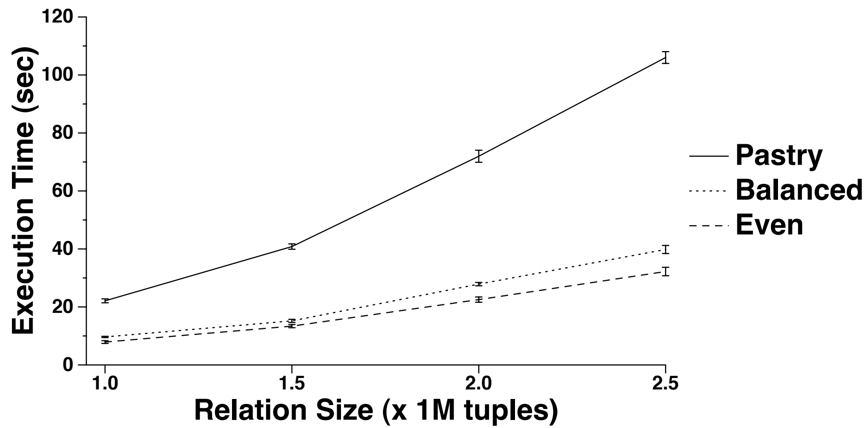
Figure 5.13: Effect of balancing on TPC-H performance. Here we use TPC-H scale factor 10 on 20 EC2 `m1.large` nodes. We show execution times for TPC-H queries Q1, Q3, Q5, Q6, and Q6, as well as the Star Schema queries 1 and 2, listed as QA and QB respectively. **Pastry** shows performance using naïve Pastry partitioning, **Balanced** shows performance using our optimized partitioning scheme (with data replication factor 5), and **Even** shows performance under our original totally even partitioning (from Chapter 4, which lacks partitioning resilience). Recall that **Balanced** can perform at best as well as **Even**, and will typically perform worse, since it is subject to limitations on data availability and the ability of the constraint solver to explore its search space.

in Figure 5.6b. It is also interesting to note that the queries that rehash more data (Q3, Q10, and the star schema queries) are more sensitive to routing imbalance; this confirms our earlier observations that repartitioning the data is an expensive operation and one that can often be a dominant cost. Fortunately, it is very parallelizable, and benefits from more even data partitioning.

We performed the same experiment on our BioJoin query and data sets. Here we explored performance on 20 of the "fast" nodes we used in the previous experiment, and also on 20 of EC2's "slow" nodes; these nodes have a single core, and much less memory (1.7GB, leaving about 1.2GB for the query processor). Figure 5.14 contains results for the fast nodes (Figure 5.14a) and slow nodes (Figure 5.14b). We see that the fast nodes are about twice as fast as the slow nodes. More interestingly, though, it is clear that the trend we saw for the TPC-H queries and the Star Schema queries holds for a chain mapping as well; optimized partitioning performs much better than the baseline Pastry partitioning, and within a small fraction of the even partitioning from Chapter 4.



(a) Fast nodes



(b) Slow nodes

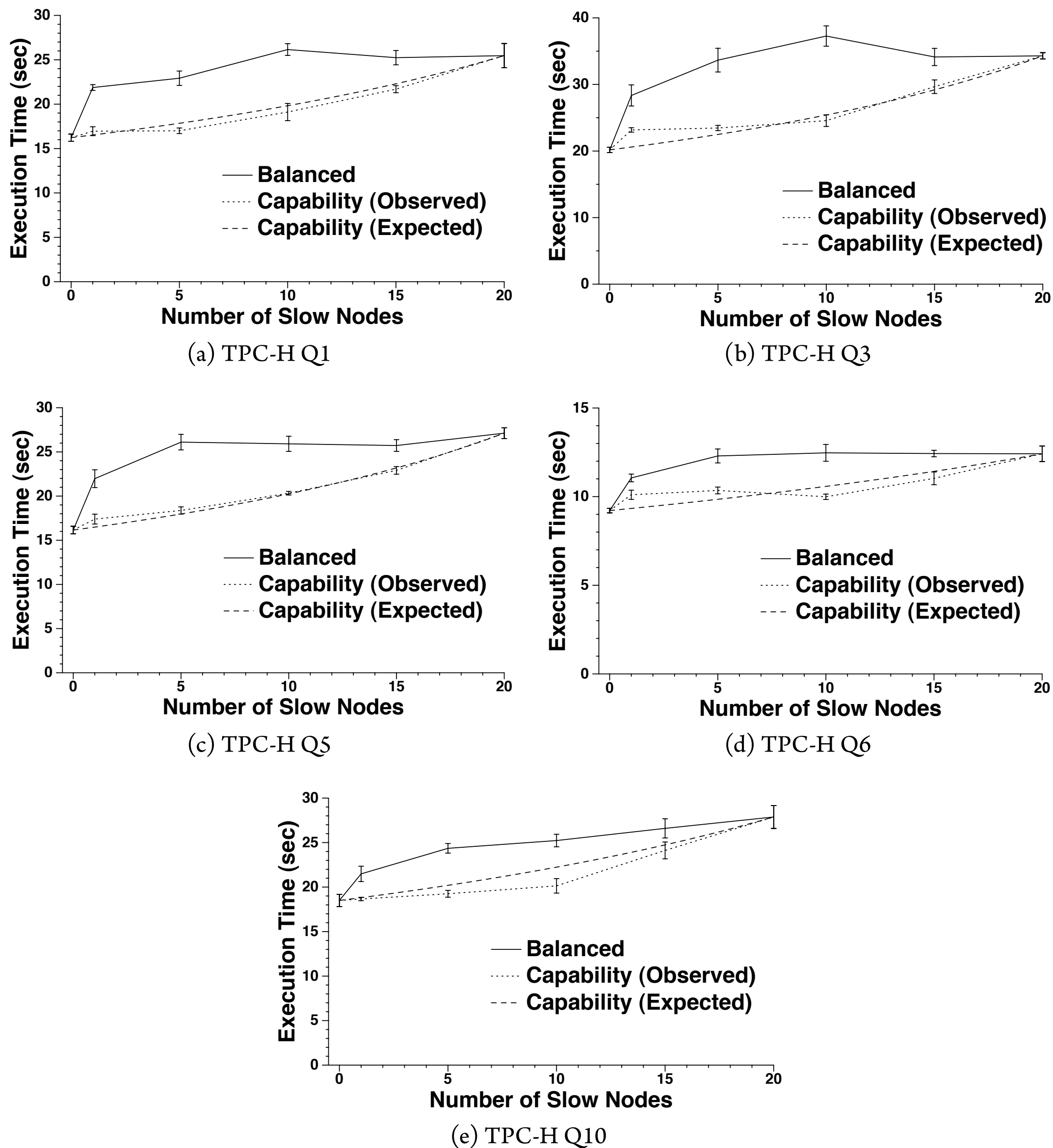
Figure 5.14: Effect of balancing on BioJoin performance. Here we explore performance of the BioJoin query at various data sizes on 20 fast (EC2's `m1.large`) and 20 slow (EC2's `m1.small`) nodes. **Pastry** shows performance using naïve Pastry partitioning, **Balanced** shows performance using our partitioning optimization scheme (with data replication factor 5), and **Even** shows performance under our original totally even partitioning scheme (from Chapter 4, which lacks partitioning resilience).

For all of the classes of queries we study, we conclude that constrained optimization to reduce routing imbalance is both feasible and effective. For modestly sized systems and reasonable replication factors, a constraint solver can create, in a reasonable amount of time, partitionings that are much better than arise from traditional partitioning techniques (like Pastry), given our desire to avoid virtual nodes. Query performance approaches that of totally even partitioning, while maintaining the reduced effects of churn offered by Pastry's partitioning resiliency.

Performance on Heterogeneous Nodes

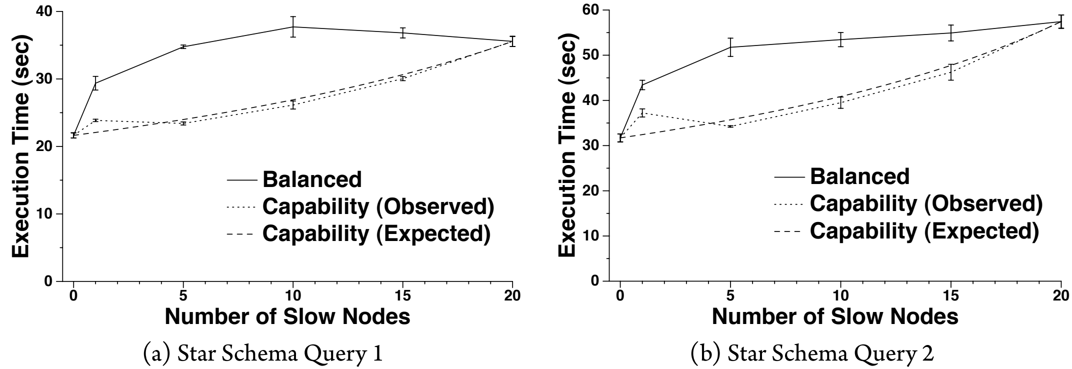
We continued our experimental evaluation by exploring performance on heterogeneous nodes. For the experiments over the TPC-H data, we used 20 of EC2's "large" nodes. We would have liked to use the less powerful "small" nodes (the only instances less powerful than the "large" nodes), as we did in the previous section. However, they have only 1.7GB RAM, leaving less that 1.2GB for query processing use. This is sufficient for the BioJoin database, as we showed in Figure 5.14b. The small nodes do not, however, have enough memory to hold 1/20 of the data and intermediate state in memory for the TPC-H data and queries. This is necessary, since disk performance on EC2 is quite unpredictable and does not give repeatable results. Therefore, we use entirely "large" nodes and slow some of them down by running a higher-priority task that iterates through a loop continuously; this monopolizes one of the two virtual CPU cores, halving the amount of processing power dedicated to our (heavily multi-threaded) query execution engine. For node heterogeneity experiments over the BioJoin data, we do use a mix of large and small nodes.

Figure 5.15 shows performance of the TPC-H queries as we vary the number of slow nodes from 0 to 20. It compares basic optimized partitioning (the **Balanced** lines) with weighted partitioning optimization (the **Capability (Observed)** lines) that attempts to assign twice as much load to the faster nodes as to the slowed nodes; in an actual deployment, the weighting can be determined through stored calibration information, which is also necessary for the query optimizer described in Section 4.4. We do not compare against the baseline Pastry partitioning or totally even partitioning, since earlier experiments established that they are not appropriate for this setting. As we see, for all queries, even one slower node causes a significant decrease in performance when the key space is distributed evenly (as that node becomes a bottleneck), and then performance degrades slightly as more nodes are slowed. Conversely, when weighting by capability, a few slow nodes slow query



(a) TPC-H Q1
(b) TPC-H Q3
(c) TPC-H Q5
(d) TPC-H Q6
(e) TPC-H Q10

Figure 5.15: Effects of node heterogeneity on performance of TPC-H queries. We show the running time for the TPC-H queries running on 20 EC2 `m1.large` nodes; some of the nodes were slowed by running a higher-priority spinning task that monopolized one of each of those nodes' two cores. All queries are posed over scale factor 10 instances (10GB data). **Balanced** shows performance when the partitioning was optimized to distribute the DHT key space as evenly as possible among nodes. **Capability (Observed)** shows performance when the partitioning was altered to distribute approximately twice as much of the DHT key space to the faster nodes, while **Capability (Expected)** shows expected performance of optimal load balancing, based on the ratio of query execution time on 20 fast nodes to that on 20 slow nodes. This assumes that execution time is directly proportional to load, which is only mostly true.



(a) Star Schema Query 1

(b) Star Schema Query 2

Figure 5.16: Effects of node heterogeneity on performance of star schema queries. We show the running time for the star schema queries running on 20 EC2 `m1.large` nodes; some of the nodes were slowed by running a higher-priority spinning task that monopolized one of each of those nodes' two cores. All queries are posed over scale factor 10 instances (10GB data). **Balanced** shows performance when the partitioning was optimized to distribute the DHT key space as evenly as possible among nodes, while **Capability (Observed)** shows performance when the partitioning was altered to distribute approximately twice as much of the DHT key space to the faster nodes. **Capability (Expected)** shows expected performance of optimal load balancing, based on the ratio of query execution time on 20 fast nodes to that on 20 slow nodes. This assumes that execution time is directly proportional to load, which is only mostly true.

execution only slightly; performance degrades slowly and predictably from when ¼ of the nodes are slowed to when they are all slowed. Figure 5.16 shows that the same trends occur for the Star Schema queries.

We also compare observed performance with expected execution cost, assuming perfect load balancing to weight key space assignment exactly in proportion to node capability. We show this using the **Capability (Expected)** lines. We derive capabilities on a per-query basis using the ratio of execution time on fast nodes to execution time on slow nodes. Of course, while per-node execution time depends on load, it only does so in a mostly linear way, and there is some overhead; additionally, optimization does not have complete flexibility to assign key space ranges, and is constrained by partitioning granularity. Nevertheless, we see that **Capability (Expected)** and **Capability (Observed)** are very close, reassuring us both of the accuracy of the cost model used for load balancing and that the optimization problem generally has enough flexibility to weight the partitioning as needed.

Figure 5.17 shows the results of a similar experiment for four different sizes of the BioJoin schema and query; here, as mentioned previously, we were able to use a mix of "small" and "large" EC2 nodes. We again see that unweighted (balanced) optimization partitioning performance degrades quickly

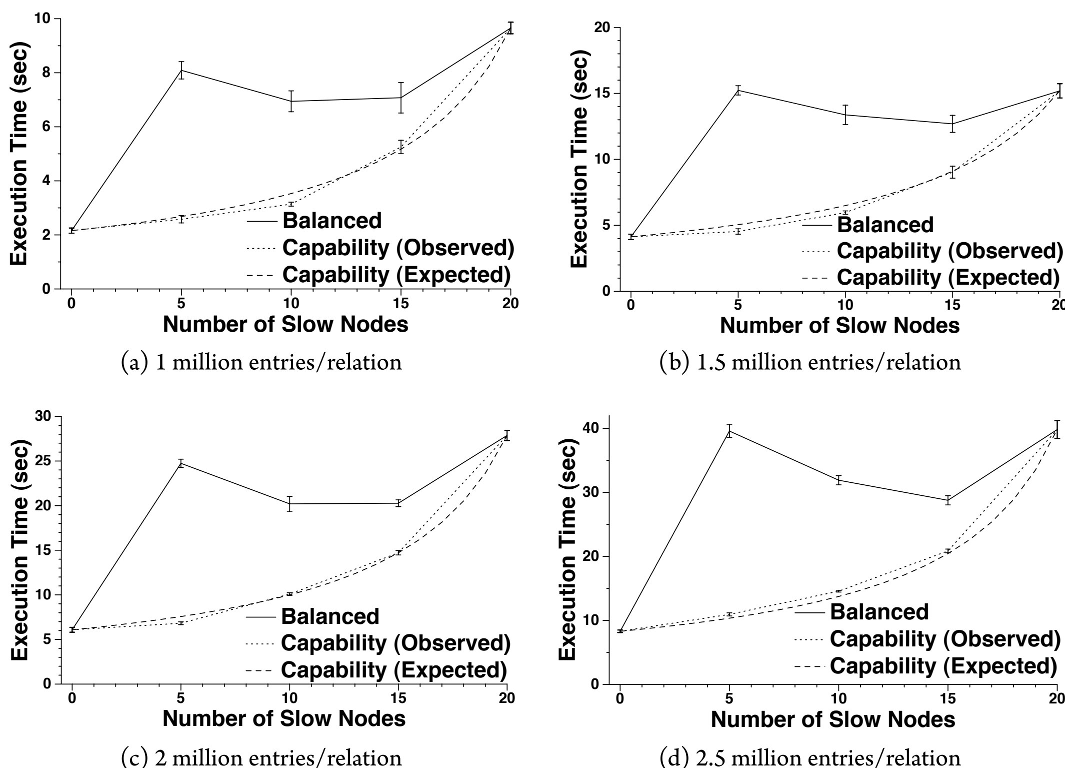


Figure 5.17: Effects of node heterogeneity on performance of the BioJoin query. We show the running time for the TPC-H queries running on 20 EC2 nodes, a mixture of faster `m1.large` nodes and slower `m1.small` nodes. We varied the size of each of the main relations (all except for the Species table) from 1 million each to 2.5 million each. **Balanced** shows performance when the partitioning was optimized to distribute the DHT key space as evenly as possible among nodes, while **Capability (Observed)** shows performance when the partitioning was altered to distribute approximately twice as much of the DHT key space to the faster nodes. **Capability (Expected)** shows expected performance of optimal load balancing, based on the ratio of query execution time on 20 fast nodes to that on 20 slow nodes. This assumes that execution time is directly proportional to load, which is only mostly true.

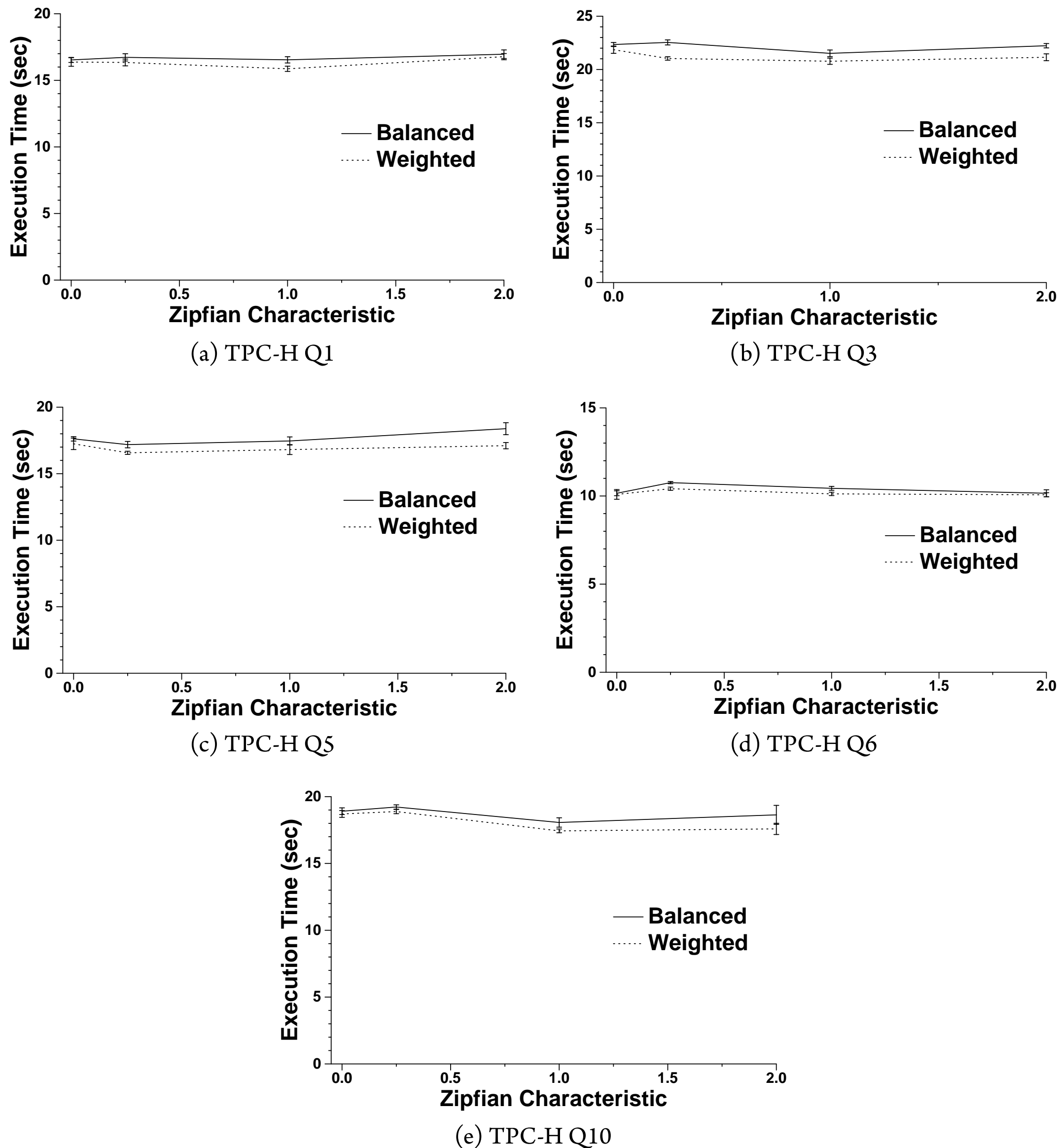
when a few nodes are replaced with slow ones; with weighted partitioning performance degrades slowly until ½ of the nodes are slow, and then more quickly until they are all slow. The extreme variation in performance of the balanced case is due the fact that the balanced partitioning for 5 nodes happened to assign several of the slower nodes more than their fair share of the key space. As before, we see that observed performance with weighting by capability is very close to expected performance if the system is able to weight the partitioning perfectly by capability.

As shown here, for a variety of workloads, weighting optimized partitioning by node capability leads to significant performance improvements. Even a single slow node can dramatically reduce performance with unweighted partitioning optimization, and slow nodes can cause odd slowdowns in query execution. Weighted partitionings, on the other hand, are predictable; they degrade gracefully as the fraction of slower nodes increases. We feel that capability-based weighting is a key addition to partitioning optimization that greatly increases its utility and flexibility.

Query Performance over Skewed Data

Our final series of experiments explores the benefits of taking data skew into account in optimized partitioning. Recall that we use the skewed data generator of Chaudhuri and Narasayya (1999), which chooses values for the TPC-H data according to a Zipfian distribution. While we expect hashing to deal with skew at least reasonably well, we were concerned about the potential for very uneven data distribution to introduce hotspots, such as, for example, the occasional very large order or very order-heavy customer. It may also cause the optimizer's derived histograms to be even more inaccurate that usual. We vary the characteristic of the Zipfian distribution from 0 (uniform) to 2 (quite skewed).

Recall that the optimized partitioning constraint problem attempts to assign chunks of the key space to different nodes. For non-skewed data, we simply divide the DHT key space into uniformly-sized chunks. For skewed data, we take advantage of the fact that our system attempts to keep each index page approximately the same size; therefore the partitioning of the key space into index pages for a relation is an approximately equi-depth histogram. We create chunks that have (approximately) the same number of index pages in them, and use those chunks for balancing purposes. As a side effect, this ensures that no index page is split between different nodes; we do not expect this to have a large effect, given that the number of index pages is typically much larger than the number of nodes,



(a) TPC-H Q1

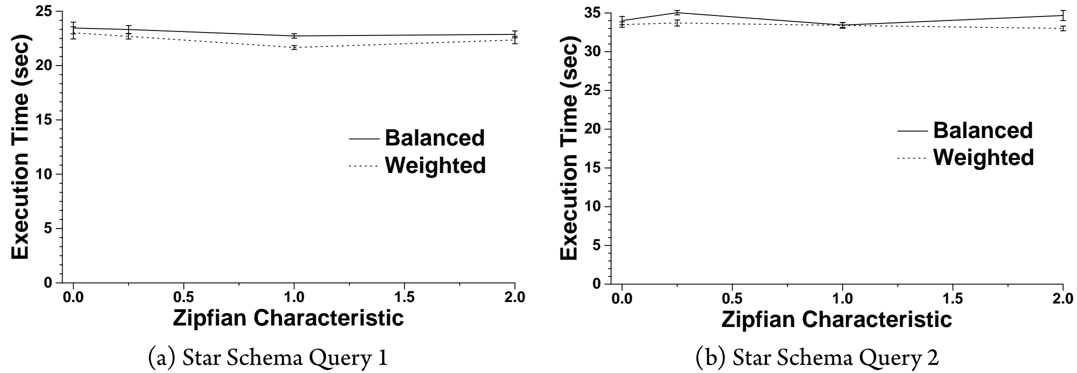
(b) TPC-H Q3

(c) TPC-H Q5

(d) TPC-H Q6

(e) TPC-H Q10

Figure 5.18: Effects of data skew on performance of TPC-H queries. We show the running time for the TPC-H queries running on 20 EC2 `m1.large` nodes. All queries are posed over scale factor 10 instances (10GB data). Skew varies from uniform (Zipfian characteristic 0) to very skewed (Zipfian characteristic 2). **Balanced** shows performance when the partitioning was optimized to distribute the DHT key space as evenly as possible among nodes, while **Weighted** shows performance when the partitioning was optimized to distribute the `LINEITEM` table as evenly as possible.



(a) Star Schema Query 1

(b) Star Schema Query 2

Figure 5.19: Effects of data skew on performance of star schema queries. We show the running time for the TPC-H queries running on 20 EC2 `m1.large` nodes. All queries are posed over scale factor 10 instances (10GB data). Skew varies from uniform (Zipfian characteristic 0) to very skewed (Zipfian characteristic 2). **Balanced** shows performance when the partitioning was optimized to distribute the DHT key space as evenly as possible among nodes, while **Weighted** shows performance when the partitioning was optimized to distribute the `LINEITEM` table as evenly as possible.

making the overhead of index page splitting relatively insignificant.

Figure 5.18 shows performance of the TPC-H queries (executing on 20 "fast" nodes, scale factor 10 as before) as we vary the skew factor. We compare normal optimized partitioning (the **Balanced** line) with optimization taking skew into account (the **Weighted** line) using the aforementioned method of assigning more weight to data-heavy regions of the key space. Recall that we can only weight the key space partitioning for one relation, due to the constraint that a single partitioning of the DHT key space be used for the entire query. Therefore we weight for the LINEITEM table, as it is by far the largest relation, and the only revelation used by Q1 and Q6. As we see, weighting has little to no effect for uniformly distributed data, and a small but statistically significant effect for most queries for more skewed data. Figure 5.19 shows that the same is true for the Star Schema queries.

From these experiments, we conclude that, at least for this data set, taking data skew into account during query execution is not necessary. The hash function does a relatively good job of handling data skew without any intervention. While it is easy to add to an implementation, the technique described here is rather limited, since it can only bias the partitioning for a single relation. Given its limited utility, and the fact that the problems we were concerned about (very large numbers of tuples with the same partitioning key, such as a the aforementioned very large order or a very order-heavy customer) seem to be more pathological than realistic, compensation for skew seems not to be a critical feature of an implementation. Certainly its effects are at best very incremental when compared to the general

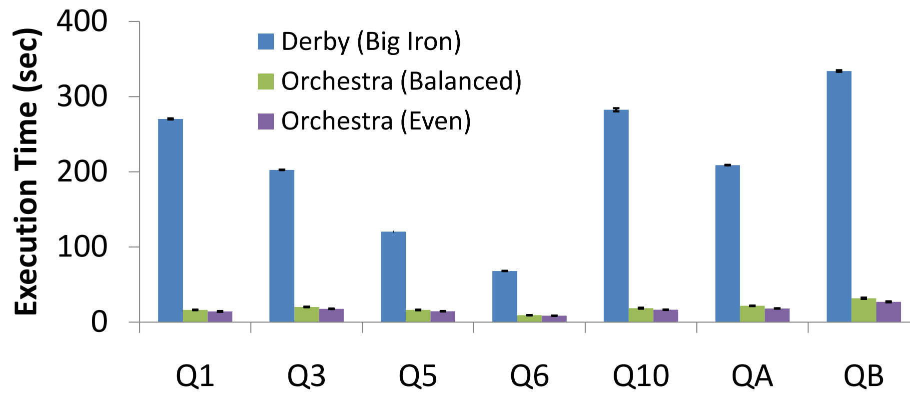


Figure 5.20: Performance comparison with Derby. Orchestra (Balanced) showed optimized partitioning to balance load as evenly as possible given data placement, and Orchestra (Even) uses totally even partitioning, which lacks partitioning resiliency. We use an EC2 `m2.4xlarge` instance, with 68GB RAM and 8 cores for Derby, ORCHESTRA uses 20 `m1.large` nodes with 2 cores, as before. Data is TPC-H scale factor 10.

partitioning optimization and capability-based weighting we explored earlier in this section. It would be interesting to explore ways to compensate for skew in multiple relations, but that would require extensive modification to the query execution (and optimization) implementation.

Comparison against RDBMSs

Finally, we also wanted to explore the overhead of our networking layer, as well as the overall quality of our implementation, by comparing against several reference RDBMS implementations. We selected Apache Derby[3] (formerly IBM Cloudscape) to compare against, as it is well-known and mature Java-based RDBMS; it therefore suffers from the same garbage collection and JIT overhead as the Java-based ORCHESTRA query processor.

We compared the TPC-H queries and the Star Schema queries against two Derby configurations. In both cases, Derby was able to fit the entire work set of tables in main memory, as was the case for ORCHESTRA. First, we compared query performance for TPC-H scale factor 10 on an extremely powerful server rented from EC2 relative to the ORCHESTRA performance we saw before, for both balanced optimized partitioning and the totally even partitioning from Chapter 4. Given that our queries parallelize relatively well, as shown previously, and that all data fits into RAM, we expect Derby performance to be limited by CPU (Derby will not use more than a single core for any given

[3] `http://db.apache.org/derby/`

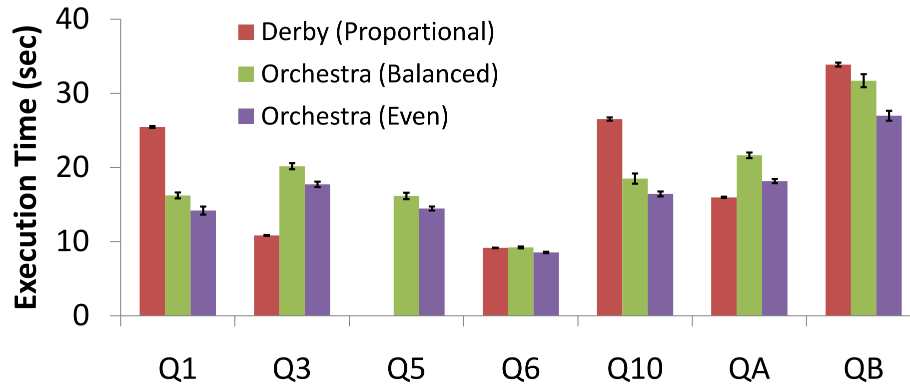


Figure 5.21: Performance comparison with Derby, proportional data. Orchestra (Balanced) showed optimized partitioning to balance load as evenly as possible given data placement, and Orchestra (Even) uses totally even partitioning, which lacks partitioning resiliency. All machines are EC2 `m1.large`. Derby uses one machine with TPC-H scale factor 0.5, ORCHESTRA uses 20 nodes, TPC-H scale factor 10. The Derby results are for 1/20 the amount of data as for ORCHESTRA, therefore giving an idea of per-node execution speed. We omit Derby results for Q5, where poor plan choice lead to an execution time of approximately 100 seconds.

query) and memory bandwidth. Figure 5.20 confirms that the greater CPU, cache, and memory bandwidth resources of our distributed approach enable it to perform better than Derby on a single powerful server; this continues to be the case, even if one takes into account Derby's greater potential for inter-query parallelism.

Next, we set out to study the quality of our distributed query execution layer and the amount of overhead introduced by the reliable versioned storage layer. Here we use TPC-H scale factor 0.5 for Derby, but continue with 20 nodes for ORCHESTRA; all nodes were identical. The Derby node therefore has approximately 1/20 the amount of data as the ORCHESTRA nodes, and therefore the same amount of data as one ORCHESTRA node, assuming even distribution. We then compare query execution times for the Derby node with this proportional amount of data with ORCHESTRA. Since all joins are foreign-key joins, with complexity linear in their input size, this scaling is valid; in the more general case, it would not be. Figure 5.21 shows the results of our proportional data experiment. We see that Derby performance is close to that of ORCHESTRA. Since execution times on a per-node basis are close to that of a mature RDBMS, we conclude that the storage overhead is not too high, and that the query execution engine performs well.

Finally, we compared against a major commercial commercial RDBMS, also running on an EC2 node. As before, the entire database could fit into the RDBMS buffer pool. The RDBMS was unable

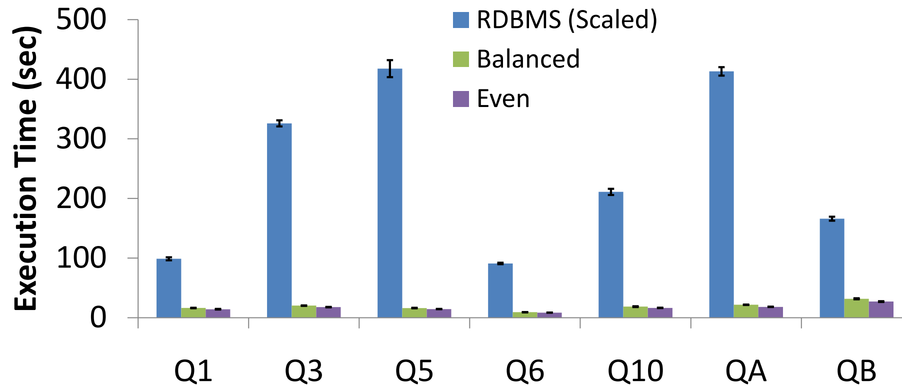


Figure 5.22: Performance comparison with commercial RDBMS. Orchestra (Balanced) showed optimized partitioning to balance load as evenly as possible given data placement, and Orchestra (Even) uses totally even partitioning, which lacks partitioning resiliency. RDBMS execution on a EC2 node with 8GB RAM and 8 cores, ORCHESTRA used 20 dual-core nodes as before. ORCHESTRA uses TPC-H scale factor 10. RDBMS used TPC-H scale factor 4, and results are scaled by 2.5 (10/4).

to load the entire scale factor 10 database, so we present results for scale factor 4, which are then scaled by a factor of 2.5; this should give a *lower bound* on RDBMS performance. This approach suffers from the same limitations on cache, CPU, and memory bandwidth as our comparison with Derby. As shown in Figure 5.22, the RDBMS performance is not good relative to distributed ORCHESTRA. It is possible that the RDBMS was not optimally tuned, as the author of this thesis is far from an expert database administrator. However, the typical users of ORCHESTRA will not be expert DBAs either, and a key point of the distributed ORCHESTRA implementation is that it is self-configuring and self-tuning. Therefore, we feel that a comparison with "out of the box" performance of an RDBMS on a large server is fair.

From both of these experiments, we conclude that, for large data sets, ORCHESTRA performance is very good. It would be interesting to compare performance on other workloads and data sizes. As shown in Chapter 4, ORCHESTRA performance on smaller workloads is generally very good. For the complex queries over large data sets studied here, performance exceeds that of large servers. It would be interesting to compare against confederations of RDBMS instances, as would likely occur in an enterprise setting; we lack the resources to do this.

5.4 Conclusions

While the techniques of the previous chapter provided excellent load balancing, they violated partitioning resilience. Any node arrival or departure would cause *all* nodes to have to move some data around to restore the desired failover properties. In this chapter, we showed how we could use traditional replicated Pastry-style data placement to avoid this, and then exploit the partitioning flexibility afforded us by the replication to improve query processing performance. We showed that this problem can be expressed in a manner solvable by a standalone constraint solver as a constrained optimization problem. We showed how this solution could be extended to compensate for node heterogeneity and data skew. We experimentally verified that an open-source constraint solver can create high-quality partitionings quickly, and that these optimized partitionings offer good performance on both homogeneous and heterogeneous networks. We also showed that data skew is not a significant problem for our system, and that taking it into account when optimizing therefore offers only limited benefit. Finally, we verified that Orchestra performance for large data sets is quite competitive with existing RDBMSs.

Chapter 6

Related Work

No work is done in a vacuum, and this thesis is by no means an exception. As hinted at in earlier parts of this thesis, many (if not most!) of the methods we used are based on, inspired by, or complementary to prior work from the literature. In this chapter, I present a broad survey of related work. In particular, I focus on how the Orchestra context differs from prior work, and how the techniques used to implement Orchestra depart from previous approaches. I begin with a comparison with prior data transformation and integration work in Section 6.1. Section 6.2 discusses issues of consistency in distributed systems. Section 6.3 reviews related work on distributed query processing. Finally, Section 6.4 discussed approaches to load balancing in distributed hash tables.

6.1 Data Transformation and Integration

One of the key tasks in Orchestra is translation between data formats. The most basic approach to translation between data formats is to write custom programs that operate on individual files, typically in some high-level scripting language like Perl or Python. Which this approach is very common, it suffers from several important drawbacks. One is that automated reasoning about the effects of entire programs is very difficult. Another is that, in most cases, separate programs must be written for each direction of translation, even though reciprocal operations are being performed[1]. A third is that much of the complexity in such programs is related to parsing and formatting, not the actual transformation. If the data is coming from a database, going to one, or both, this is needless complexity;

[1] Foster (2009) and Foster and Pierce (2008) explore ways of eliminating this problem for certain classes of transformations.

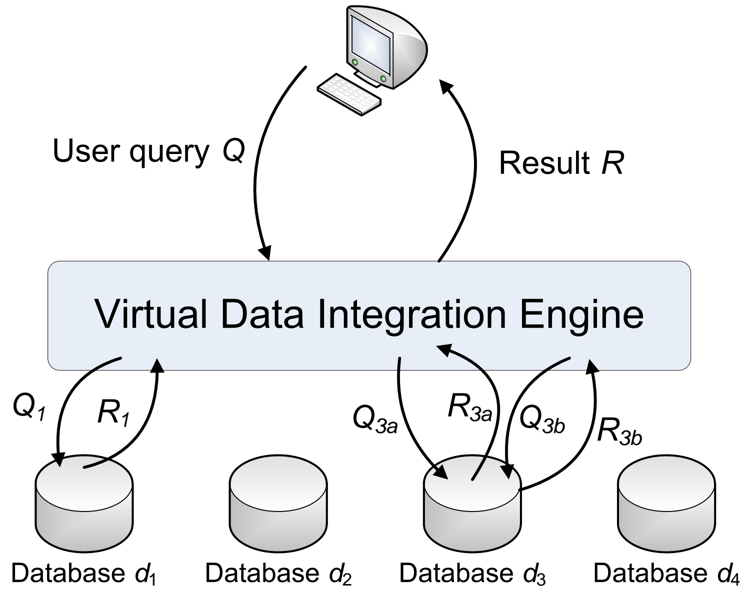


Figure 6.1: Virtual data integration. A user query Q is rewritten into queries Q_1, Q_{3a}, and Q_{3b}. The system was able to determine that source databases d_2 and d_4 do not have data relevant to Q. Q_1, Q_{3a}, and Q_{3b} are posed over d_1 and d_3. The system then combines R_1, R_{3a}, and R_{3b} into the overall result R, and returns it to the user.

one can get around this final limitation by using a database access layer like JDBC or ODBC along with the general-purpose programming language. However, the other problems remain.

Given limitations of encoding data transformation between databases in general-purpose programming languages, it is not surprising that *data integration* (as it is known in the academic community) and *enterprise information integration* (as it tends to be called commercially) has been an active area of research and development. This work is closely related to Orchestra, in that it enables queries to be posed over a different schema than the schema (or schemas) that hold the primary copies of the data. One common feature of all these systems is that they allow the relationships between tables in databases to be specified as *mappings* in a high-level, declarative fashion. Research systems typically use mappings in some variant of Datalog (Ullman, 1988, chap. 3), as does Orchestra; commercial systems use a variety of techniques. The details of how the mappings are expressed is not important for this discussion.

Data integration systems can be partitioned along several axes, including where the data is queried, how the relationships between databases are specified, and whether all querying must be done over a single schema. We now explore these variations.

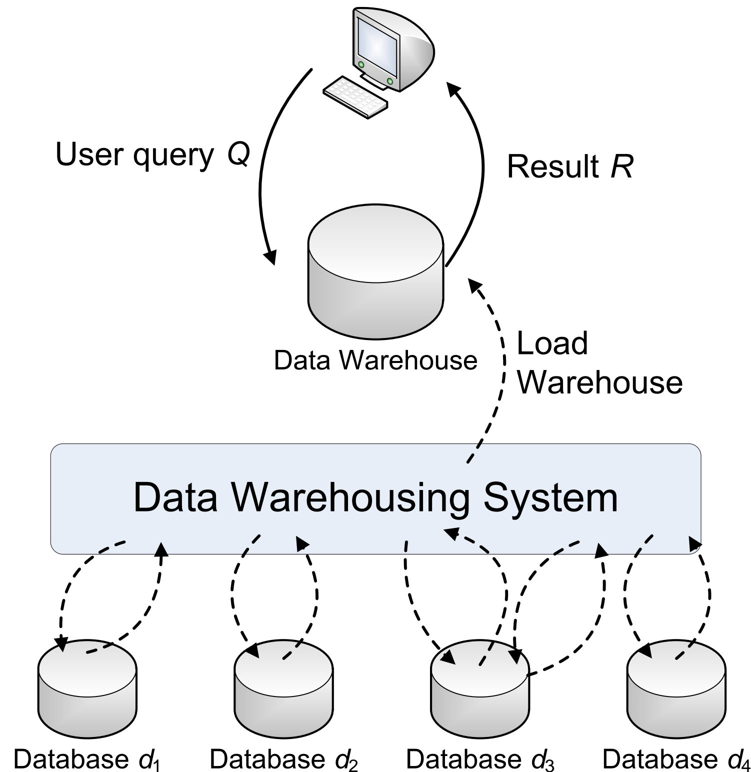


Figure 6.2: Data warehousing. Periodically, the system builds a data warehouse by posing queries over the source databases d_1, d_2, d_3, d_4; this is shown with dashed arrows. Later, a user poses a query Q; this is answered directly over the stored data warehouse.

Logical and Physical Data Integration

We begin by exploring variation in data placement, and where data is queried. Some systems perform what is known as *virtual data integration*. In this method, data never leaves the original databases (the source databases) that own the data. Figure 6.1 shows a virtual data integration system in action. A user query is to the the system (sometimes referred to as a mediator), using a schema supplied by the system. When there is only one such schema, as here, this is called the *global schema* or *mediated schema*. The mediator then rewrites the query over the mediated schema into queries over the source databases. The results of these queries are combined by the mediator (possibly in some complex fashion, using a join or aggregation, or possibly simply by concatenating the results), which returns them to the user. In *data warehousing* (sometimes called *physical data integration*), the source databases are used periodically to build (or later, to refresh) a data warehouse, a database which holds a copy of the relevant data from the sources, translated into the mediated schema. Later, a user poses a query

to the system. It is directly executed over the database holding the data warehouse, and the results are returned to the user. Figure 6.2 shows the relationship between a data warehousing system, the source databases, and an end user posing a query. Chaudhuri and Dayal (1997) contains an overview of data warehousing that the interested reader may find useful.

Some applications are better suited to virtual data integration, and some to data warehousing. Data warehousing systems are often simpler to build, since they don't need to contain logic for query rewriting; this can prove quite complex. However, data warehousing systems suffer from stale data, if the warehouse is not refreshed enough. Also, data that is never queried is still maintained in the data warehouse, potentially at great cost. Data warehouses do, however, reduce the load on the source databases, if the query workload is high. A common application of data warehousing is for business analytics. Here, corporations unify data from many different systems (which are often used by customer-facing services, and therefore should not be subject to the high load induced by complex analytical queries) into a single large database. The complex analysis queries are then performed over that isolated database. Since the queries are used for planning, it is not necessary that the data be completely fresh; warehouses are often rebuilt or refreshed nightly or over weekends.

For applications where freshness *is* critical, virtual data integration is necessary. For example, a flight search engine would need to use virtual data integration, since it must acquire up-to-date pricing and availability information from each airline in order to respond correctly to user queries. Also, given the large number of available routes, it would not be feasible to build a data warehouse; instead, the system performs the searches to retrieve the desired routes as they are requested.

Hybrid approaches are also possible, and indeed appropriate for many settings. A data warehouse can be incomplete, and query source relations as needed. Caching can be added to virtual data integration to avoid consulting the source databases for all queries. Recent work on this topic includes, for example, Haas et al. (2010).

Mapping Specification

A second axis in data integration systems is how the mappings are specified. Let us assume for the moment that there is a single global mediated schema, as in the previous examples. There are then queries that relate that express the relationships between that schema and the source databases. Early systems, such as TSIMMIS (The Stanford-IBM Manager of Multiple Information Sources),

described in Garcia-Molina et al. (1997), express the (possibly virtual) global database as a view over the source databases. Other systems, such as the Information Manifold (Levy et al., 1996), express the source databases as views (or more accurately, as subsets of views) over the global database. These approach are known respectively as GAV (global-as-view) and LAV (local-as-view). Lenzerini (2002) gives a nice discussion of the fundamental differences between the two approaches, from a theoretical perspective, and lists many more systems that use the two approaches. In the context of data warehousing, mappings are also be expressed in a general-purpose programming language; this approach is known as *extract, transform, and load*, or ETL. However, as discussed in the introduction to this section, it is difficult to reason about the effects of whole programs, and therefore such mappings cannot be used for query reformulation in virtual data integration. We do not consider such approaches further in this chapter.

Let us consider an example similar to that from Levy et al. (1996). We have four databases, one containing used cars for sale, another containing luxury cars (new cars costing at least $50,000) for sale, another containing vintage cars (from before 1950) for sale, and another containing motorcycles for sale. We also have a database that gives a description of features for each make, model, and year. Suppose the schemas were, respectively, as follows:

UsedCars `(make,model,year,price)`

LuxuryCars `(make,model,year,price)`

VintageCars `(make,model,year,price)`

Motorcycles `(make,model,year,price,condition)`

Features `(make,model,year,features)`

The attributes are all as one might expect, and *condition* is always either 'new' or 'used'. Suppose the global schema consists is **Vehicles** `(make,model,year,price,condition,features)`.

In Datalog, one might express the *Vehicles* relation using GAV mappings as

```
Vehicles(mk,mdl,yr,prc,'used',feats)   :-  UsedCars(mk,mdl,yr,prc),
                                           Features(mk,mdl,yr,feats)
Vehicles(mk,mdl,yr,prc,'new',feats)    :-  LuxuryCars(mk,mdl,yr,prc),
                                           Features(mk,mdl,yr,feats)
Vehicles(mk,mdl,yr,prc,'used',feats)   :-  VintageCars(mk,mdl,yr,prc),
                                           Features(mk,mdl,yr,feats)
Vehicles(mk,mdl,yr,prc,cond,feats)     :-  Motorcycles(mk,mdl,yr,prc,cond),
                                           Features(mk,mdl,yr,feats)
```

Recall that in Datalog, the different rules represent different ways to derive tuples in a relation, and so this Datalog program takes the union of these simple transformations of the source relations. It is interesting to note that some of the information from the source database descriptions has been lost. While it is apparent from the GAV specification that *VintageCars* only contains used cars, it is no longer recorded that they must be from before 1950. Similarly, for *LuxuryCars*, while it is recorded that they are all new, it has been lost that they all cost at least $50,000.

Using conjunctive queries, one might express the relationships between *Vehicles* and the sources using LAV mappings as

UsedCars(make,model,year,price)	$\subseteq$	*Vehicles(make,model,year,price,'used',_)*
LuxuryCars(make,model,year,price)	$\subseteq$	*Vehicles(make,model,year,price,'new',_)*, $price \geq 50000$
VintageCars(make,model,year,price)	$\subseteq$	*Vehicles(make,model,year,price,'used',_)*, $year < 1950$
Motorcycles(make,model,year,price,condition)	$\subseteq$	*Vehicles(make,model,year,price,condition,_)*
Features(make,model,year,features)	$\subseteq$	*Vehicles(make,model,year,_,_,features)*

These are containments, not equalities, since some tuples in the global schema could come from multiple relations; for example, a used car from 1947 could be either in *UsedCars* or *VintageCars*. While LAV is perhaps conceptually more difficult than GAV (since the common operation in a data integration system is to take the union of the sources, it makes sense to think of the specification that way as well), it is more powerful, since it contains additional information. Furthermore, that additional information is helpful for query reformulation; a query for new vehicles need only search the tables *Motorcycles* and *LuxuryCars*.

The third, and most general way to express the relationship between the mediated schema and the sources is using GLAV (global/local-as-view), introduced by Friedman et al. (1999). As they observe, pure LAV and pure GAV suffer from problems that did not become obvious in the relatively straightforward example above:

> Using either pure LAV or pure GAV source descriptions has undesirable consequences. In LAV, the mediated schema must contain all attributes shared by multiple source relations, whether or not they are of interest in the integration application ...To make matters worse, some sites use shared attributes that are only meaningful internally, such as URLs of intermediate pages or local record ids. In GAV, on the other hand, the mediated schema relations must all be relations present in the sources, or conjunctive queries over them, making the mediated schema contingent on which source relations are available.

GLAV allows more complex transformations to be present on both sides of the mappings. These are

equivalent in power to *tuple-generating dependencies*, or TGDs (Fagin et al., 2005), which are used in many modern data integration systems, including ORCHESTRA. TGDs have the form

$$\forall \overline{x}, \overline{y} \, (\phi(\overline{x}, \overline{y}) \Rightarrow \exists \overline{z} \, \psi(\overline{x}, \overline{z}))$$

where ϕ is a conjunction of relational atoms over the source relations, and ψ is one over the mediated schema (or other target relation). While more complex to consider for the purposes of query reformulation, the increased expressive power and flexibility of GLAV/TGDs finds them in use in much recent theoretical data integration work, such as notion of *data exchange* developed by Fagin et al. (2005). The *certain answers* query semantics of this approach are also adopted by ORCHESTRA. TGDs are also used as the basis for the Clio data integration and data exchange system (Fagin et al., 2009).

Peer Data Management

The systems presented above are all traditional data integration engines, where there is a single mediated schema. All mappings directly relate source databases to the mediated schema, and it is over this mediated schema that all queries are posed. While appropriate for many settings, this approach has some serious shortcomings. It is difficult to add new data to the mediated schema (all mappings must be updated), and users that don't have the flexibility to use a schema they prefer (they must use the mediated schema). Additionally, it can be difficult to represent (though perhaps possible with the extensive use of null values) data that is only present in some of the sources.

Peer data management systems (PDMSs), such as Piazza (Halevy et al., 2003) and Hyperion (Kementsietsidis et al., 2003), and the more theoretical work of Calvanese et al. (2004), address this by removing the constraint of a single mediated schema. Each peer in the system has its own schema, and that schema is directly connected to only some of the other participants' schemas. Queries can be posed over any schema, and will return both data form that schema and data translated into that schema. Because the system has an an interconnected "web" of mappings, data may have to travel across a chain of mappings in order to travel from the source to the query; there may even be multiple paths between a source and the query schema, which return different results. Figure 6.3 shows an example of the relationships between schemas in Piazza. In such a system, it is easy to add (or remove) participants at any time, and to introduce new mappings to supplement the existing relationships between peers. In systems that perform virtual data integration, such as Piazza, query reformulation

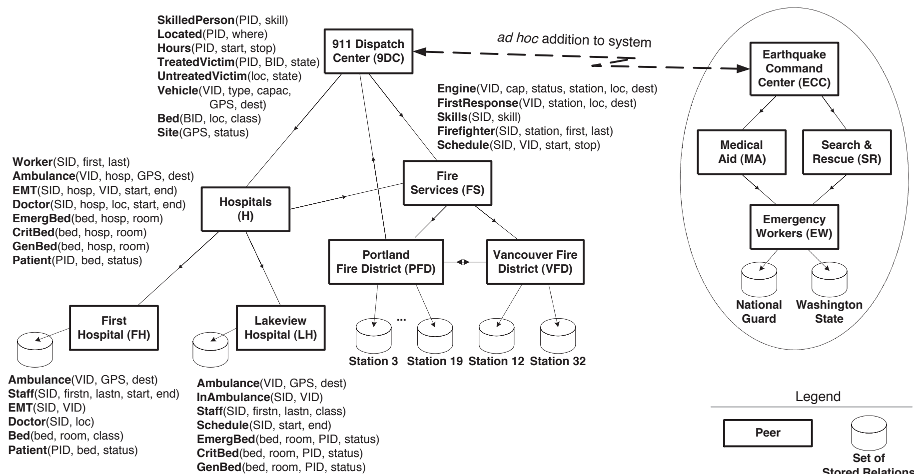


Figure 6.3: Schema mappings in Piazza. Observe that there is no global schema, and the mappings are specified between any two schemas in the system. Translating data from one schema to another may require the composition of many mappings, and there may be multiple such compositions available to use. This approach was the inspiration for ORCHESTRA. Figure is taken from Halevy et al. (2003).

becomes extremely involved, and in any such system care must be taken to avoid mapping cycles that, which coupled with existentially quantified variables (to indicate missing data), can create infinitely large instances. PDMSs, as they are known, are undoubtedly the most complex of the systems presented here. However, their flexibility and ability to evolve make them a compelling choice for many dynamic, *ad hoc* applications.

Relationship to ORCHESTRA

ORCHESTRA builds upon the data integration ideas and techniques presented here. Its logical model is very close to that of Piazza. ORCHESTRA is precisely the sort of *ad hoc* application where the PDMS approach excels. We do not want to burden the initial ORCHESTRA participants with the task of coming up with the perfect mediated schema, which will be adequate for the lifetime of the collaboration; it may not yet be clear what form the collaboration will take. The PDMS-style web of mappings also allows existing end user applications to continue to work over their current schemas unchanged, and allows new participants to join easily by creating mappings to a participant with a similar structure. If these mappings are not sufficient, they can be supplemented later; the initial amount of effort required to join, however, is as low as possible.

As mentioned previously, ORCHESTRA uses GLAV mappings, in the form of TGDs, to express the

relationships between participant schemas. Unlike Piazza and the Information Manifold, however, it is not a query reformulation engine, and does not perform virtual data integration. Orchestra maintains a (custom) local instance for each participant (as in physical data integration), for reasons of privacy and query performance. This also allows querying to take place offline, or when some other participants are not available. In this sense, Orchestra comes close to maintaining a data warehouse for each participant.

Orchestra is not just about translating data between schemas, though that is a key functionality. None of the data integration systems discussed here place much (if any) emphasis on data consistency or integrity. The goal is to get as much data as possible into the query schema (at least virtually), and then to answer queries over it; the end user is responsible for removing dirty, incorrect, or stale data. A major focus in Orchestra is being more selective about the data that is brought into a user's instance, to eliminate unneeded or untrusted data, and to maintain consistency. We address related work on this topic in the next section.

6.2 Consistency and Data Sharing

Once data is translated between different schemas, there is often a deeper source of incompatibility. There will typically be some *inconsistency*, as the different parties may have different levels of data freshness, some sites' data may be dirty, or, frequently, the sites may have different *viewpoints* about what data is factual or what the precise meaning of the data is. Data consistency is mostly orthogonal to data integration and translation between schemas. In Orchestra, as described earlier in this thesis, we perform the translation through update exchange, and then ensure consistency and integrity through the reconciliation process. Reconciliation happens entirely in the participant's own schema. Here we present related work on data consistency in several contexts. It is worth bearing in mind that most of the approaches here assume that a consistent global instance is possible; this is a fundamental mismatch with the goals of Orchestra.

Data Cleaning

There has been a variety of work on *data cleaning* or "repair" schemes, which take a set of constraints and a data instance that may violate some of these constraints. Data cleaning techniques used in data warehousing typically rewrite the database to satisfy the constraints (and may propagate corrections

back to the source databases). Rahm and Do (2000) presents a survey of such techniques. More theoretical work, such as that presented in Arenas et al. (1999) and Lembo et al. (2002) focuses on rewriting queries to give the results that would have resulted from a consistent data instance.

These techniques of course assume that a consistent global instance is possible and indeed desirable. They are very good at correcting errors where one or a few databases have stale data; for example, they will easily fix a situation where one of a company's many systems has an out-of-date address for an employee. These types of problems arise frequently in data warehousing. However, if there is a large amount of conflicting data, it is not clear *a priori* which alternatives should be preferred; metrics like edit distance work less well.

One approach to consistency in ORCHESTRA would be to create possibly inconsistent instances, like those that arise from a PDMS, and then apply a cleaning technique to restore integrity constraints. However, we wish to account for differing levels of trust; perhaps the payroll system is more authoritative, so if its address is different, it is because the employee recently moved and other systems, such as those for health insurance and the pension plan, have not yet been updated. We also wish to take into account relationships between data items; if the employee's address update was in the same transaction as the deletion of the employee from a list of out-of-state residents, we should ensure that that update survives as well. While some of these problems can be solved in the data cleaning framework, we wish to make extensive use of metadata, like provenance and transactional dependencies, to avoid making potentially arbitrary decisions between consistent alternatives. We therefore need a more complex approach than that offered by data cleaning.

Distributed Replication

The context of ORCHESTRA is similar to that of replica management, version control, and disconnected filesystems. Such systems create a local *working copy* for a user, which can then be read and updated offline. The user may then later connect with the rest of the system, which may be a central server or more complex distributed system, to share their changes with other users, and to receive changes from other users. This is clearly very close to ORCHESTRA's periodic publish and reconciliation operations. Unlike in ORCHESTRA, however, there is generally a focus on maintaining a single, master, consistent instance; in ORCHESTRA, user instances are intentionally allowed to diverge. In other related fields, such as distributed databases, it is at least hoped that most operations will take

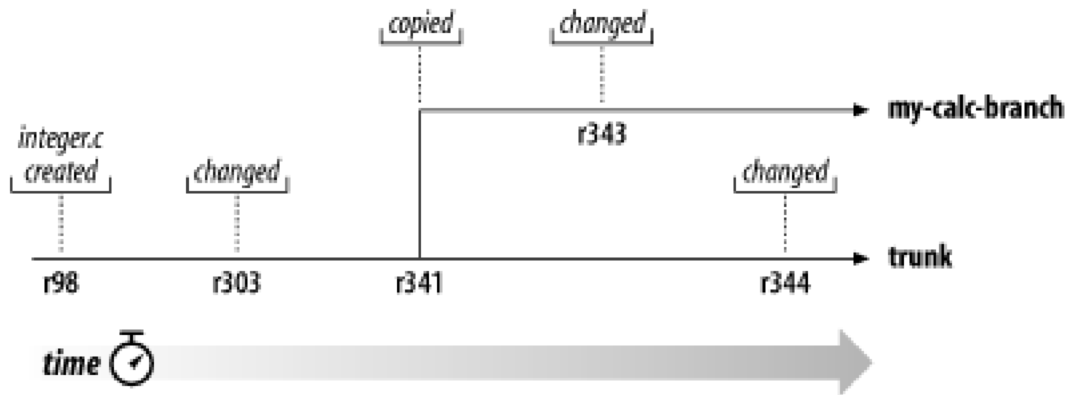


Figure 6.4: Branching in a version control system. Figure is taken from Pilato et al. (2008).

place online; however, they fall back on a working copy model similar to that described above to enable work to continue during a network partition, and then effectively publish their local changes when connectivity is restored.

Version Control

Client/server version control systems are widely used in academia and software development. Well-known and widely-used such systems include the open-source CVS[2] and Subversion[3] systems, as well as the proprietary Perforce[4] system. Such systems allow for both diverging and non-diverging replicas. The default mode of operation in such systems is to require all user-supplied changes to be specified against the current (commonly referred to as HEAD) revision of files in the system. If a user has made changes against an older version of the files, they must update their local copy to the HEAD version before they can commit; this may entail manually resolving their local edits with recently made edits by others. Version control systems will attempt to do this automatically for edits to different files, and different regions of files; however, the end user is ultimately responsible for making sure that the changes they made to their local copy still make sense against the HEAD revision. Dealing with "merge conflicts," as they are known, and solving problems caused by incorrectly performed merges, are common headaches for software developers. It is not uncommon for a user to inadvertently undo the effects of other recent changes when committing theirs. Version control systems also allow divergent replicas, not on a per-user basis, but in the form of "branches" of the

[2]http://www.nongnu.org/cvs/
[3]http://subversion.tigris.org/
[4]http://www.perforce.com/

files they hold. The system may create a copy of a particular version of the files stored, and use this as the basis of the working copies for different sets of users. Figure 6.4 shows how a branch is created. Once the system has branched, however, it is typically very difficult to merge one branch into another; while the branches have a common ancestor, they are treated independently by the system from the moment of divergence onwards. The branching can be localized to a particular portion of data in the system, typically using the filesystem-style hierarchy that organizes the files under version control.

A key difference from ORCHESTRA is that all clients must commit changes relative to the most recent version of the system state. If a user has made incompatible changes, he or she cannot commit those to the system without resolving the conflict (excluding the possibility of branching). This can be very burdensome, and cause users to commit less frequently; this increases the potential for data loss as the user's work is not backed up on the server. In ORCHESTRA, on the other hand, publishing always succeeds; conflict resolution is put off until the user's local instance is updated. The analogous operation in version control is to create a separate branch for each user; if no one else updates that branch, then their commit (publish) operations always succeed with no merge conflicts. However, other users do not continue to receive the *compatible subset* of the changes the user makes, and as mentioned previously, merging divergent branches of a version control system can be a very tedious task.

Distributed version control systems, such as BitKeeper[5], Git[6], and Mercurial[7], are a recent development in version control, and address some of these issues. Effectively, each user's working copy becomes its own branch. The issues of merging changes between branches, which must be done on request, still remain. While ORCHESTRA also considers each user's instance as in effect its own branch, it tracks dependencies between the updates to users instances (i.e. working copies) and has semantic information (such as integrity constraints) that allows it to effectively perform much of the merging automatically, based on user interested (as expressed by mappings and trust conditions).

[5]http://www.bitkeeper.com

[6]http://git-scm.com

[7]http://mercurial.selenic.com

File Synchronization and Distributed Filesystems

File synchronizers such as Unison (Pierce and Vouillon, 2004) take two directory structures and efficiently propagates changes between them to keep them as close to each other as possible. In addition to only performing pairwise synchronization, Union is conservative and will never overwrite changes made by a user; when a conflict exists, both versions are typically preserved and the user must resolve the conflict before continuing to work.

The Boomerang bidirectional programming language (Foster, 2009) assumes that only one of the "source" or "view" is editing between synchronization, and so conflicts do not arise in this context; the focus is on translation. The related Harmony project (Foster et al., 2007), a synchronizer for tree-structured data (perhaps translated into a tree representation by Boomerang), does consider conflicting updates to nodes in a tree. They discuss the trade-offs between *persistence*, where no changes made are ever undone, with *convergence*, where the replicas do not totally converge. The authors of Foster et al. (2007) state (emphasis original):

> In Harmony, we have chosen to favor persistence because it is easier to ensure that unsupervised reconciliations are safe. (Unsupervised reconciliations are extremely desirable from the point of view of system administration, facilitating automatic reconciliation before disconnection or upon re-connection to a network, via nightly scripts, etc.) Divergent systems are more likely to allow users to proceed with their work—the set of replicas may be globally inconsistent, but it is more likely that each replica is locally consistent. By contrast, convergent systems are more likely to force a user to resolve a conflict after a *remote* user initiated a synchronization attempt. For example, consider conflicting updates to a file with strict syntax requirements (e.g., LaTeX or C); the convergent system's attempt to record both updates may result in a file that causes subsequent processing to fail.

The replicas will not converge until they are manually edited to become so, though the divergence will be localized to the subtree where the conflict occurred.

The IceCube system (Kermarrec et al., 2001) can also be thought of as a file synchronizer, though it in fact operates over operation logs rather than raw files, as Unison does. It allows many users to operate on their replicas off-line, and then periodically it reconciles the operation logs of the users. Using domain-specific information, it can consider different valid interleavings and reorderings of subsets of the combined operation log; using a domain-specific scoring function, it chooses a maximal interleaving. Some of the operations performed are rolled back, and all replicas are set to the result of performing the selected subset. This means that IceCube is a convergent system, in contrast

to Harmony above. As we have stated previously, we do not feel that convergence is an appropriate goal in ORCHESTRA: customized user instances are expected and desired, and it is unclear how a global preference between conflicting updates could be specified. Also, this approach requires all replicas to be simultaneously available for the system to perform a global reconciliation; we were eager to avoid this requirement in ORCHESTRA. Interestingly, IceCube does have an idea analogous to the flatten operation discussed in Chapter 3: it attempts to combine operations on the same data item into a single one to eliminate transient conflicts and increase the number of operations that can be applied.

Bayou (Edwards et al., 1997) is a replicated, weakly consistent storage system. It is designed to be the storage layer for distributed applications with varying or intermittent network connectivity. An application can perform a read or write operation at any replica; updates will eventually be propagated to all other replicas. This "eventual consistency" model means that semantic conflicts can arise due to replication lag. Since Bayou has no knowledge of application semantics, it depends on the user to supply *application-specific conflict detectors* and *conflict resolvers*. These are responsible for bringing replicas into an internally consistent state after the techniques of Petersen et al. (1997) transfer updates between replicas. Bayou does provide "session guarantees," described in Terry et al. (1994) which ensure that a particular instance of the application will see a consistent view of the storage layer *by constraining the replicas with which it can communicate*; they are not sufficient to impose global atomicity or serializability. Like IceCube, Bayou required the user to write custom resolvers and detectors in order to converge on a globally consistent instance. However, the IceCube interface is at a somewhat higher level, and provides built-in support for reordering and equivalence detection; such capabilities would have to be added to Bayou on a per-application basis. Regardless, Bayou's convergent properties set it apart from ORCHESTRA's goals.

Finally, conflict detection and resolution occurs in distributed filesystems during a network partition. Both more traditional network filesystems like Coda (Satyanarayanan et al., 1990), replicated filesystems like Ficus (Reiher et al., 1994), and peer-to-peer filesystems like Ivy (Muthitacharoen et al., 2002) support updates during a network failure. When connectivity is restored, divergent replicas must be identified and repaired. Since filesystems have no knowledge of the meaning of the files they hold, both Ivy, Ficus, and Coda work to detect conflicts, and then disable updates to them; application-specific conflict resolvers are then used to perform a *repair* operation by combining the various versions of a file into a unified one. Like most of the other file-based approaches discussed

here, Coda, Ivy, and Ficus are convergent.

Of the systems discussed here, only Harmony allows replicas to diverge, and attempts to keep the divergence to a minimum; while parts of replicas diverge, the rest of them is kept in sync. This approach is similar to what we intend to do in ORCHESTRA. However, our data is not tree-structured, and we wish to use additional information to resolve some conflicts automatically. The filesystem (or filesystem-like) techniques of IceCube, Bayou, Ivy, and Coda all rely on custom-coded conflict resolvers to create a globally consistent instance by discarding some incompatible updates. IceCube is in some ways the most similar to ORCHESTRA. Despite its convergent properties, it uses operations (like updates) instance of instance state as the base of it semantics. It also considers interactions between the operations, dependencies between them, and attempts to reduce transient conflicts between operations through a flattening-like procedure.

Distributed Databases

Distributed databases typically attempt to provide the illusion that all transactions execute over a single, consistent database instance. If partition occurs and operations from different sites are incompatible, the goal is to choose the subset that satisfy some sort of optimization criteria. Often this involves choosing some (i.e. the largest, or somehow the most important) subset of transactions that are in some way compatible, such as respecting serializability.

Some early systems, like earlier versions of IBM's System R*, lacked replicated data. It used variants of the standard two-phase commit protocol to ensure serializability between sites (Mohan et al., 1986). Since data was not replicated, there was no potential for replicas to diverge, and therefore no conflicts to resolve; a network partition would simply cause the system to grind to a halt.

More interesting from our perspective, and more related to conflict resolution, are systems that do have replicated data. An early example of such a system is SDD-1, as presented in Bernstein et al. (1978). In general, a distributed protocol is used to keep data replicas synchronized. Then, in the event of network partition, querying can continue, assuming there are replicas of all of the needed tables in a user's partition; it is not always possible to make updates in the event of a network partition. Davidson (1984) presents an *optimistic* approach to concurrency control in this setting, which allows general transactions to occur during a network partition. Transaction committal is held until the network partition is resolved, and the author presents a detailed comparison of the performance

of several *backout selection strategies* that restore global serializability by rolling back a subset of the transactions. Backout selection is NP-complete for even simple metrics like minimizing the number of transactions rolled back, and therefore also for more useful ones like minimizing the global back-out cost with different weights for different transactions; the author presents and compares several heuristics.

Version vectors (Parker et al., 1983) are a widely-used method of determining causality in distributed systems. For a system with n nodes, a version vector is an n-ary vector of counts $[c_1, \ldots, c_n]$. When a new data item is created, it's version vector is initialized to all zeros. When node n_i updates an item, it increments the ith entry of its version vector. As long as a distributed, replicated system remains connected and its consistency protocol is observed, all version vectors for a data item will stay synchronized. If the network is partitioned, however, the version vectors establish a partial order (the so-called "happened-before" relation) over the versions of the data item. We say that a version vector v_i precedes another v_j (i.e. $v_i < v_j$) if, for $1 \leq k \leq n$, $v_i[k] \leq v_j[k]$; this means that the data item for v_j was derived from that for v_i by applying a sequence of operations. Since version vectors establish a partial order, it is possible that neither $v_i < v_j$ or $v_j < v_i$; then some updates have been applied to one data item that have not been applied to the other, and vice versa. This means that consistency has broken down, perhaps due to a network partition. After this is discovered, the application will perform a reconciliation to merge the two data items somehow, and associate with the merged data item the version vector that is the pairwise maximum of each of the elements of v_i and v_j.

Version vectors are the building blocks for optimistic concurrency control and replication, which can be used to implement a variety of distributed systems where strict serializability is no longer required. Saito and Shapiro (2005) presents a survey of work on optimistic replication; distributed filesystems like Ivy, Coda, and Ficus described above use optimistic replication and version vectors, in concert with conflict resolvers, to maintain consistency. Microsoft Access also uses (or at least used at one time) vector vectors to ensure consistency between multi-master replicas of its databases (Gray et al., 1996; Hammond, 199?). It attempts to resolve conflicts indicated by the version vectors automatically, but in complex scenarios will simply store an error for human review.

Ceri et al. (1995) presents a distributed, replicated database, where updates will eventually be applied to all replicas after a partition is healed. Serializability, however, will not be enforced, transactions that read stale data will not be rolled back, and constraints may be violated. A later application-

specific cleanup might be necessary to deal with, say, overdrawn bank accounts or overbooked flights; however, as in Davidson (1984), the authors feel the ability to get useful work during a partition is worth the cost of some cleanup (or transaction rollbacks in Davidson (1984)). Ceri et al. (1995) uses version vectors to determine which updates need to be sent between sites during the reconciliation phase after a partition heals. In some ways, their consistency model is very close to ours, in that they (have the option to) ensure a write ordering on updates, as does ORCHESTRA. ORCHESTRA, however, does not require eventual consistency, which they do.

Approaches from distributed databases are not generally applicable in the ORCHESTRA context. Much of the distributed database literature is concerned with strict serializability over a global instance; since ORCHESTRA lacks a consistent global instance, serializability has little meaning in our context. Version vectors, as used in Ceri et al. (1995) are somewhat related to our approach, since they ensure that updates are applied at replicas in the same order they were originally. However, we do not necessarily apply all transactions to each participant, leading to divergence from the eventual consistency that they are seeking. Since our transactional guarantees can introduce dependencies *between* data items (i.e. tuples with the same primary key), version vectors cannot be directly applied to the ORCHESTRA context. It is not clear with what a version vector would be associated.

In principle, it could make sense to assign a version vector to a particular region of the database (some set of keys in some set of tables) that has particular semantic meaning as a logical entity. The system could then use a version vector to keep track of the updates participants make to that portion of the database, and use this instead of transactional dependency model we have proposed. Each transaction would have an associated pre-condition version vector, taken from the state of the region when the transaction is created, and that transaction could only be applied to other instances with that version vector for the region of the database the transaction references. This would ensure participants end up in one of a only a few fixed states for the database region. In effect, the region of the database is treated as an opaque blob which ORCHESTRA cannot see inside; it can merely ensure that the operations on this blob are valid. However, if reconciliation is infrequent, and each participant makes a few updates to the region, the participants could totally diverge. If the database region is large, divergence could be all but assured, and if the database region is very small, then the end result will get very close to our existing transactional and conflict-based semantics. Nevertheless, it is interesting to consider the relationships between our consistency model and others from distributed systems.

6.3 Distributed Query Processing

Distributed query processing as a field dates back almost to the beginning of the relational database era. Some influential early systems, such as IBM's System R* (Lindsay et al., 1984) and the Computer Corporation of America's SDD-1 (Jr. et al., 1980), supported operations over distributed data. However, as discussed in Mackert and Lohman (1986) and Bernstein et al. (1981), both approaches consider partitioning of data at the table level. While this can afford some performance benefits, partitioning of tables among a collection of nodes can offer greater benefits through *partitioned parallelism*, which was introduced in the context of distributed Ingres (Epstein et al., 1978). By executing portions of each operation on a large number of nodes at the same time, this can offer greater data retrieval speed (due to parallel I/O) and greater processing speed (due to parallel processing). In general, there is a trade-off between the cost of network communication and the benefits of partitioned parallelism. However, network speeds (both in the local area and at the global scale) have improved dramatically over the last several decades, and now often exceed that of physical storage. Partitioned parallelism in shared-nothing databases has been cited by leading researchers (e.g. in DeWitt and Gray (1992)) as the future of high-performance databases. The major commercial RDBMSs, namely IBM's DB2, Microsoft's SQL Sever, and Oracle's eponymous RDBMS all support partitioned parallelism.

While state-of-the-art commercial RDBMSs can offer high performance, they are notoriously difficult to set up and use, even at a single site. A key goal of ORCHESTRA is to provide reliable, high-performance storage and querying without the need for a central server. Multiple computers at multiple sites will therefore fulfill the storage and execution need of ORCHESTRA. We do not wish to require complete reliability on the part of any particular site, allowing intermittent network failures, hardware upgrades, or software faults. Furthermore, we want the system to be "plug and play" in that a node needs to know the identity of another node, and connect to it; from there, everything should be self-configuring. Therefore, we chose to implement a partitioned-parallel, peer-to-peer database for ORCHESTRA.

Peer-to-Peer Storage and Query Processing

The literature contains several recent, influential peer-to-peer query processors, PIER (Chun et al., 2004; Loo et al., 2004; Huebsch et al., 2005; Huebsch, 2008) and Seaweed (Narayanan et al., 2008;

Mortier et al., 2006). Both use the distributed hash table (DHT) as their data partitioning layer, which we first review. Well-known DHTs include Pastry (Rowstron and Druschel, 2001), CAN (Ratnasamy et al., 2001), Chord (Stoica et al., 2001), P-Grid (Aberer, 2001), and Bamboo (Rhea et al., 2004). While the details of distributed hash tables vary from implementation to implementation, they all implement one key feature: given a data item, they can associate it with a node. Both keys and nodes are given IDs in some large key space; this is often 160-bit integers, chosen to match the output of the SHA-1 hash function. Hashing is used to ensure an even distribution of nodes and data items throughout the keyspace; data items are associated with the hash of their key, and nodes (typically) with hash of their IP address. The key space is partitioned in a deterministic way using the node IDs; the exact method depends on the system. Such systems cannot guarantee complete partitioning (i.e. routing) consistency under churn, though they typically guarantee that routing will stabilize once churn has ceased.

The key operation in a DHT is to send a message to the node that owns a particular DHT ID[8]; the message might be to store a retrieve a data item from persistent store, if the system is implementing a "pure" distributed hash table that just holds data, or it might do something more complicated. Message routing is accomplished by consulting the *routing table* at each node, which indicates which a request should be sent to for certain regions of the key space. To enable scalability to very large numbers of numbers of nodes, i.e. to Internet-scale systems, multi-hop routing is used. Each node has incomplete knowledge of the nodes in the system, and so many nodes (typically logarithmic in the system size) may be contacted as a message makes its way to the node that owns a particular data item. Note that in ORCHESTRA we using single-hop routing, which is possible due to the smaller system size. We then use snapshots of a single consistent routing table for the duration of a query, which guarantees consistent partitioning for the purposes of that query.

Distributed hash table-based query processors have largely targeted dynamic and loosely consistent, rather than static and strongly consistent, data. Huebsch (2008), for example, discusses applying PIER to keyword search in peer-to-peer filesharing, and to network monitoring. PIER is build over the Bamboo DHT mentioned previously. It supports storing data natively in the DHT; this is never the primary copy of data, as nodes must periodically publish data to the DHT. This ensures

[8]This is a slight generalization. Some DHTs include persistent storage as a built-in feature, meaning that the basic operation is simply to store or retrieve a data item. Others, such as Pastry, only focus on routing; this is more flexible, since it enables high-level applications to be built over the DHT. We assume a Pastry-style interface.

reasonable data freshness, that data moves to the correct node as the nodes come and go (there is no background replication), and that data from nodes that have left the system does not persist indefinitely. Data may also be stored in application-managed storage on the nodes, and pushed into PIER for processing during query execution. PIER supports partitioned-parallel operations, such as joins, by using the DHT as a very large hash table for pipelined hash joins; it also has optimizations to reduce network traffic by using Bloom joins (Mackert and Lohman, 1986), and by caching in concert with the multi-hop routing inherent in the DHT to produce results early. It also supports pre-computation of aggregate results at each node (into a *partial state record*, or PSR), and using tree to combine the PSRs incrementally until they reach the node that issued the query.

Some of the techniques used by PIER inspired approaches in ORCHESTRA: we use similar join techniques, though without any caching (since we have single-hop routing), and also use PSRs. We do not use trees to merge PSRs, however. Since we have summary statistics, we can determine the number of PSRs the query issuer is likely to receive. If receiving and combining them is determined (by the optimizer, in a cost-based fashion) not to be too heavy a burden, the query issuer will simply combine all the PSRs itself. Otherwise, the PSRs will be rehashed, and a distributed partitioned-parallel aggregation will be performed to produce a complete results for each aggregate group; these are then sent to the issuer. Our approach works well if the number of nodes is relatively small (as is typically the case in ORCHESTRA) or if the number of aggregation groups is very large (rehashing will cause even load distribution). If a large number of nodes produce results for only a few groups, the incremental aggregation approach of PIER will perform better. In ORCHESTRA, however, we have not found aggregation performance to be a bottleneck.

Like PIER, Seaweed focuses on distributed computation of aggregates, and considers network management as a driving use case. Unlike PIER, no base data is stored in the DHT (which in this case is Pastry); the DHT is instead used to maintain information about past node availability, so that the system (and user) can make intelligent decisions about how long to wait for results. A user might be willing to wait, say, one hour for results from 95% of the nodes, but not an extra four hours to reach 99%. Like PIER, it also uses hierarchical computation of aggregates, but unlike PIER it does not support distributed joins. It does, however, place a stronger emphasis on consistency, in that it ensures that network churn will never cause available data to be skipped or processed multiple times; data from nodes that are down for the duration of the query, as chosen using the node availability information, will not be considered. PIER, in contrast, only offers best-effort consistency.

A key difference between the needs of ORCHESTRA and these prior peer-to-peer query processors is that they are focused on the scalability afforded by peer-to-peer systems; we are focused on their self-configuring nature and resilience to failure. Additionally, we need to ensure consistent access to *mutable* data. DHTs use replication to enable transparent failover by replicating data items to nodes near (in the DHT key space) to the node that owns that item; if the owning node fails, requests will get rerouted to a node that owns a backup copy. Pastry, like other DHTs, uses background replication through swapping summary structures[9] is used to ensure that each node maintains copies of the data items that it should, even as the set of nodes changes due to churn. Replication ensures that, with high probability, data is not lost, and the background process will eventually migrate it to the correct nodes. PIER does not use replication; instead, any data stored in the DHT can be through of as a cache that is refreshed periodically from master copies elsewhere. Seaweed uses Pastry replication to ensure that availability metadata is not lost, though they do not seem concerned about consistency of such metadata.

This approach causes several problems for mutable data. One is more technical, in that replication implementations (at least in FreePastry) assume that each DHT ID refers to a single, immutable data item. It will ensure that a copy of a data item with that ID is replicated to the correct nodes. If the data item (or items) a DHT ID changes over time, the system will not propagate such changes. Versioning can perhaps be added to such a system, but consistency issues may result, as in the distributed databases discussed above. A more fundamental problem is that writes may not be seen immediately, if there is a network partition or incorrect routing; this can lead to divergent replicas. Replication will eventually bring the most recent version of a data item to the correct nodes; at any point in time, however, all nodes in the system would have to be consulted to find the most recent version of a data item (or even to determine if it exists), since it might not yet have migrated.

We solve this problem in ORCHESTRA through the use of versioned storage (for tuples), a hierarchical index that maps versions of the database to versions of tuples, and a global distributed counter to keep track of versions of the database. This approach will not work for Internet-scale systems, due to the complexity of possible counter implementations. However, it means that we can use background DHT replication (with versioning, as mentioned above, of both base data and index pages) for the common case (data is correctly located) and can search on demand to retrieve the most recent

[9] See details at `https://mailman.rice.edu/pipermail/freepastry-discussion-l/2005-September/000168.html`

copy of missing or stale data; the index can be used to determine when this situation exists. It can also detect data loss of base data and index pages.

An alternate approach to this problem, described in DeCandia et al. (2007), is taken by Amazon's Dynamo storage engine. Like DHTs, it uses hashing for data distribution, replicates data (to a configurable N nodes) for reliability, and focuses on high update availability in the event of node failure. Data is versioned using vector clocks (very similar to the version vectors presented previously). Updates are always specified relative to a particular version. The node that owns a data item acts as the coordinator and master copy for reads and writes of that data item; note that which node is the master may change due to node failure or partition. In a write, the coordinator tries to update all N replicas, and must wait for a certain number of them (configurable W) to commit the write before it can return success to the user. Similarly, in a read the coordinator reads as least a configurable R replicas, and returns the most recent (using their versions). If replicas have diverged, application specific replication occurs. By tuning R and W relative to N and each other, Dynamo can favor read or write performance, though perhaps at the expense of delayed inconsistency detection. While this approach is interesting and clearly high-performance, we felt that for Orchestra the potential for stale reads was too high.

In short, Orchestra's query engine is inspired by prior DHT-based query engines. Unlike them, however, we provide stronger consistency guarantees, and use the DHT for the primary storage of base data. Given our focus on peer-to-peer system's reliability and fault tolerance rather than scalability, we were able to exploit our smaller scale to provide guarantees that they cannot. Here we have outlined related work on query processing and consistent storage in peer-to-peer systems. In the next section we describe related work on ensuring reliability in distributed query processing.

Reliable Distributed Query Processing

Reliable distributed query processing is a topic of study dating back to IBM's R* (Lindsay et al., 1984) and SDD-1 (Bernstein et al., 1978), and perhaps best known commercially as Tandem NonStop SQL (Tandem Database Group, 1987). However, their consistency model and definition of reliability differ somewhat from ours. In NonStop SQL, the problem is detecting a failed machine in a local cluster and possibly aborting and restarting a query. As mentioned in an Section 6.2, in R* and SDD-1, the focus is on ensuring transaction serializability. Aborting a transaction (which may con-

tain both queries and updates) and restarting it is both necessary and expected in the event of node or link failure.

Our consistency model is relaxed, as we do do not consider transactions (when publishing the update log, we do consider them during reconciliation) or serializability. We simply need correct and complete answers to a query relative to a particular version of the database. Churn could cause loss or inconsistent partitioning of intermediate state, and therefore missing or incorrect results. For us, node arrival is not a problem due to the use of per-query routing snapshots, but node departure causes data loss. One option is to abort and restart the query; reliable storage will ensure correct results. We would like, however, to avoid redundant work and to incrementally recompute "missing" answers where possible, in order to complete query computation. In Section 4.4, we described how we can use buffered state and replicated data to recover the state at failed nodes, avoiding much recomputation when recovering from node failure.

In many distributed systems, there is no incremental recovery of query results. What we call "abort and restart" is the only recovery mode. Some do support techniques at least similar to ours. The OGSA-DQP system (Smith et al., 2000, 2002), a grid-based distributed query processor, is perhaps the closest to our setting, though it is not peer-to-peer in nature. OGSA-DQP was extended in Smith and Watson (2005) to support recovery from node failure during query execution through the use of *upstream backup*. This technique ensures that copies of tuples are buffered as they are sent, and retained until they (or all of the tuples derived from them) have flowed through some number of downstream nodes; in this way, a small number of node failures will never result in loss of query results. It has limitations when processing joins, as one relation must be completely buffered; the memory cost of this is no higher, however, than using pipelined hash joins, as we do. The main difference from our work is that buffering is explicit. We instead exploit existing buffers, and rescan base data only when existing buffers are not sufficient to recover needed state.

Recent work on reliable stream processing systems has also lead to related techniques for reliable query processing. Reliability is if anything even more critical in stream processing, as restarting a query will loose any state accumulated in intermediate stream operators. Hwang et al. (2005) gives a nice comparison (albeit using a simulation for experimental results) of different techniques for building robust stream processors. Upstream backup is considered in this context as well, and performs well if failures are rare as it does not add much processing overhead or network traffic; however, it can take a long time to process the backed up data in order to recover the state of a failed node. Other

techniques considered include *active standby*, where multiple copies a query are run simultaneously on different nodes. This allows for a failure to be quickly corrected, but seriously reduces the available processing power and adds considerable network overhead, as all computation is performed multiple times. Balazinska et al. (2008) details the use of active standby in the Borealis stream processor. It is also used, via the Flux operator (Shah, 2004) in the TelegraphCQ system (Chandrasekaran et al., 2003). *Passive standby* is a promising hybrid approach, which combines some amount of upstream backup with periodic checkpointing of operator state to a backup. If the amount of state is smaller than the amount of input used to generate it (as it common for windowed and aggregate operators), this can lead to a dramatic reduction in network traffic while causing an acceptable increase in time needed to recover from a node failure. Hwang et al. (2007) and Kwan et al. (2008) discuss ways to add passive standby to Borealis, and offer promising results. These methods have higher network overhead than ours, since redundant data (in the form of operator checkpoints or duplicate computation) is sent over the network. Additionally, we can try to exploit existing buffers to reduce the cost of recovery. Since we can always rescan base data, it doesn't matter that some needed data may not be buffered anywhere; for stream processing, of course, rescanning base data is not an option.

Cloud Computing

Our target domain is more controlled and smaller than Internet-scale peer-to-peer networks. In many ways, our work is closer to cloud data management platforms. Yahoo!, for example, is developing the PNUTS (Cooper et al., 2008) massively parallel database, together with the "Pig Latin" query language (Olston et al., 2008), which will allow complex queries in the style of MapReduce (Dean and Ghemawat, 2004, 2010) to be specified more easily. Like our work, PNUTS provides strong consistency guarantees and supports transparent failover. MapReduce-style workloads (PNUTS uses the Hadoop MapReduce implementation) offer support for restart of failed work units, and are highly partitioned-parallel. While most of the features of our work are present in PNUTS, there are a few major differences: PNUTS depends on a reliable message broker to achieve consistency, and does not support consistency *between* records. While global consistency is important for Orchestra, global consistency in (say) the Yahoo! user database is not necessary; the authors of PNUTS instead opt for higher performance through the use of per-item master copies. Additionally, MapReduce typically depends on a single, reliable controller.

Google's MapReduce implementation can also be used to perform complex transformations over distributed data stored using Google's internal systems, such as the GFS distributed filesystem Ghemawat et al. (2003), or the Bigtable distributed database (Chang et al., 2008). Such systems also offer automatic load balancing and failover, like ORCHESTRA. However, they require a permanent controller, or root node, and also do not offer system-wide versioning consistency (Bigtable and GFS only support row- and file-level consistency). MapReduce is also quite low-level for use as a query language; by comparison, the query layer for ORCHESTRA uses standard SQL. The Pig Latin query language, mentioned above, is one example of a project to provide a higher-level (though not as high as SQL) query language for MapReduce-type transformations. The HadoopDB academic research project (Abouzeid et al., 2009) is also attempting to provide SQL-style querying over Hadoop, the open source MapReduce implementation. While it provides good query performance, and uses load balancing to operate over a heterogeneous cloud, the work presented so far focuses on query performance for analytical workloads; there is no discussion of a consistency model. Additionally, like other MapReduce implementations, Hadoop depends on the central *NameNode* to store filesystem metadata; HadoopDB therefore suffers from a single point of failure. In a related projected, Facebook has built Hive Thusoo et al. (2010), a Hadoop-based data warehouse. Its emphasis is also on query performance over very large data sets, and not consistency.

The ORCHESTRA query processor and storage layer shares many features with cloud-based databases: these include fault tolerance, massively parallel processing, use of heterogeneous commodity hardware, and relaxed consistency semantics. ORCHESTRA, however, has requirements that the cloud systems lack. We want it to be totally automatically configuring; while cloud systems typically support transparent failover, they require initial setup. We also want stronger consistency semantics. While we are not interested in full serializability, as many traditional distributed databases are, we do desire globally consistent versioned snapshots; cloud approaches usually offer only record- or tuple-level consistency. Finally, many of the cloud approaches have a permanent central controller or master node. Our substrate, in contrast, uses peer-to-peer storage, and query initiator as a *temporary* master node just for the purposes of query execution.

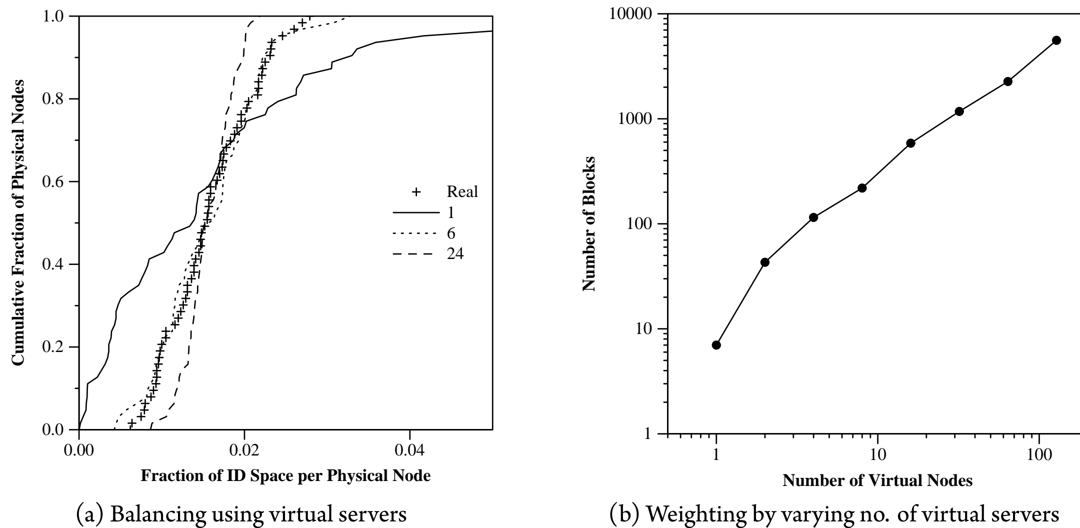


Figure 6.5: Virtual servers in CFS. Figure 6.5a shows the fraction of the ID space assigned to each of 64 physical nodes using from 1 to 24 virtual nodes at each physical node. The "Real" line shows the distribution of 10,000 data items between to physical nodes for the 6 virtual node case; it is very close to the key space distribution for that case. Note that this differs from the graphs shown in Chapter 5 in that our histograms showed how loaded the *most loaded* node was over many instances; this shows how loaded *each* node is for a particular instance. Totally even load balancing would assign each node $\frac{1}{64} = 0.016$ of the key space. Figure 6.5b shows, for seven nodes with numbers of virtual nodes varying from 1 to 128 (i.e. 255 virtual nodes total), how many of the 10,000 data items (filesystem blocks in this case) are assigned of those nodes, as a function of the number of virtual servers. The relationship is approximately linear, making it easy to assign more load to more capable servers. Figures are taken from Dabek et al. (2001).

6.4 Load Balancing in Distributed Hash Tables

There are a number of approaches to load balancing in peer-to-peer networks. The most basic load balancing task is simply to allocate the network's key space evenly among all nodes; in a naïve DHT the distribution of range sizes among nodes is roughly Poisson (Ledlie and Seltzer, 2005). *Virtual servers* were first introduced in a different context by Karger et al. (1997). Dabek et al. (2001) applied virtual servers to DHTs to improve the performance of the Cooperative File System (CFS), a peer-to-peer filesystem built over the aforementioned Chord (Stoica et al., 2001) DHT. Each node runs a large (often 5 to 20) virtual servers. Each virtual server is assigned its own node ID, and therefore gets its own region of the key space. While the amount of key space at each virtual server is highly variable, since each node runs a large number of them, the total fraction of the key space (and therefore the total amount of data) at each node is very consistent. Figure 6.5a shows a histogram of the fraction

of the key space assigned to each physical node for a variety of numbers of virtual nodes. As the number of virtual nodes increases, the distribution of the key space (and data items) to physical nodes becomes more consistent.

However, even distribution of key space among physical nodes does not always offer optimal performance. If the nodes are heterogeneous, it is often desirable to allocate more of the key space to the more capable nodes. This virtual server approach of CFS can be used to assign more data data to such nodes, by creating more virtual servers at more powerful nodes. As Figure 6.5b, the number of data items at each node is directly proportional to the number of virtual servers there.

A different approach, described in Surana et al. (2006), also takes differing node capabilities into account but goes beyond this, considering which data items are more popular and attempts to distribute the key space to keep each node operating at a reasonable fraction of its capabilities. This technique must also factor in the cost of moving state between nodes. The techniques proposed for Orchestra are similar, but there are important differences in the setting. In Orchestra, the query workload is such that extreme hot spots, like the so-called "Britney Spears problem" in peer-to-peer filesharing (the nodes which own those two keys are swamped by teenyboppers looking for a copy of her latest hit), are unlikely; each node's load is instead strongly related to its fraction of the DHT key space. Their approach attempts to move large chunks of data between servers in a cost-based way; due to replicated data, we have the routing flexibility to move *the retrieval* of smaller chunks of data from node to node at little to no cost.

In general, creating more virtual servers increases the evenness of key space distribution among the physical nodes. However, as discussed in Ledlie and Seltzer (2005), Karger and Ruhl (2004), and elsewhere, virtual servers also have some negative effects:

1. They increase churn. Each node has more neighbors, so more nodes acquire new regions of the key space when a node arrives or departs.

2. They increase the amount of routing state, and therefore also the cost of performing each routing step. They also increase the amount of bandwidth used for network maintenance.

3. They increase the number of hops per lookup, in DHTs that perform multi-hop routing. Recall that Orchestra does not.

4. They complicate replication, since care must be taken to ensure that data is not replicated at another virtual server on the same node.

For Orchestra, there are several additional reasons that virtual servers do not work well. By increasing fragmentation of the key space, they decrease the benefit of clustering index pages near (in the DHT key space) to the tuples they reference: this increases the number of tuple IDs sent over the network, and can cause a major increase in network traffic and decrease in performance in bandwidth-constrained settings. Additionally, it takes a very large number of virtual nodes to approach totally even partitioning; in a partitioned-parallel query processor like Orchestra, the most loaded node is likely to be a bottleneck, and a smaller number of virtual servers often leave a few nodes with substantially more data that others. We validated experimentally that virtual servers do not work well in our setting; performance was several times worse than with the approach shown in Chapter 4.

For Orchestra, we wanted to use a load balancing approach that did not significantly complicate the existing implementation, but offered better replication properties (decreased data churn at node arrival and departure) than totally even key space assignment, as used in Chapter 4. As discussed in Chapter 5, we switched to Pastry-style range assignment, and exploited the routing flexibility afforded to us by data replication to improve performance. Since our system already supported routing table snapshots for per-query consistency, the changes needed were minimal. Though not discussed in this thesis, since we are only changing where data is *used*, not where data is *available*, it is also possible to use different partitionings for different queries.

There are other approaches that do not use virtual servers at all, or use them in a more complex way to get greater benefit from using fewer of them, mitigating some of their shortcomings described above. We here review some of these approaches, many of which could be complementary to our existing Orchestra implementation. We did not study them experimentally, since their implementation could be quite complex, but they are a promising direction for future research.

The approaches of Karger and Ruhl (2004) and Ledlie and Seltzer (2005) are similar in that they attempt to create only a few virtual nodes per participant, but they choose the nodes' DHT IDs from a small set of alternatives. Karger and Ruhl (2004) attempts to evenly distribute the key space as much as possible, or alternatively the number of items at each node, by switching between one of a few possible node IDs. This does cause increased churn when nodes arrive and depart, as $O(\log \log n)$ nodes may change their ID when any node comes or goes; caching can ameliorate the data migra-

tion cost, however, as a node may already have copies of some of the items it is now responsible for. The approach of Ledlie and Seltzer (2005) is more general, as multiple virtual servers may be used at each physical node, but again their IDs are not fixed; more virtual servers may be added, as needed, to keep the utilization of the nodes in the system (they focus on network bandwidth) within some range. Sampling is used to determine the expected effect of different virtual server ID choice. Other approaches to key space assignment include the threshold approach of Ganesan et al. (2004), which attempts to keep the utilization of all nodes within some fixed ratio, and the more probabilistic approach of Manku (2004).

There are also approaches to optimizing performance in DHTs without altering the key space distribution. Since they do not consistently distribute the key space among nodes, they cannot be used as the basis for a partitioned-parallel query processor like Orchestra. The approach of Dabek et al. (2004), for example, optimizes read performance by storing redundant copies of (relatively large) data items and using the results that are returned most quickly. They either replicate entire data items (like most DHTs) or use erasure coding to store some number of partial copies of a data item, any sufficiently large subset of which can be used to to reconstruct it. They do not consider either write performance or the consistency issues that would arise from a "write-anywhere" model that would likely accompany this approach.

Chapter 7

Future Work

There are a number of promising directions in which to take the work presented in this thesis. I break these down into three categories, corresponding to the ideas presented in Chapters 3, 4, and 5, respectively.

7.1 The CDSS Model

There many possible variations of the CDSS model that it might be interesting to explore. We discussed in Chapter 3 the idea of adding read dependencies to transactions. It would be interesting to fully characterize the effects this would have both on the semantics of consistency, and on the rate of divergence of participants in a CDSS. It would also be interesting to explore more deeply the effects of update exchange on the semantics of transactional dependency. If a tuple is already in a participant's instance, and another derivation becomes available, that participant's instance does not change. However, if that participant then makes an update to that tuple, what is the antecedent of that transaction. Is it either, or both? The single-schema setting of Chapter 3 did not consider this and related questions.

There many simpler tweaks that can be made to Orchestra. For example, while we require all updates in a transaction to be trusted for the transaction to be trusted, we assign a transaction the priority of its highest-priority update. This favors high-priority updates, but only if the entire transaction is trusted. It would be interesting to explore other variants on this, such as using the lowest priority update in a transaction, or no longer requiring that the entire transaction be trusted. It

would be interesting to see the results of a user study, showing which semantics real users, as opposed to database researchers, find most useful or explicable.

Aside from simple user studies, large-scale use of ORCHESTRA in a real-world context is clearly the next step. The theory behind collaborative data sharing is well understood, and the implementation of the system is now stable and reliable. It would be interesting to see how real scientists, business-people, or government agencies make use of collaborative data sharing. They may have needs that we have not anticipated, or come up with unforeseen, pathological use cases that cause performance degradation.

7.2 Distributed Storage and Querying

The distributed storage and query engine presented here has grown from a relatively simple prototype to a large and complex piece of software. As with any piece of software, once it has reached some level of maturity, it is worth stepping back and considering various design decisions, and seeing if they continue to make sense.

At a relatively superficial level, it would be worth reimplementing the system in C++ rather than Java. The run-time overhead imposed by the JIT and garbage collection is quite high, and made timing-based experiments somewhat tricky, since results took a while to converge. Additionally, it proved necessary to change the query execution layer to operate over large blocks of raw data, instead of structured Java objects, for performance reasons. At this point, there is relatively little benefit in terms of code simplicity from using a somewhat higher-level language like Java.

I would also like to implement query execution operators that work well with persistent storage. While to some extent one of the benefits of massively parallel computation is the abundance of RAM and the greatly reduced need to go to disk, it would be nice if join and aggregate operators could neatly serialize their state to disk, if memory became constrained, rather than relying on the operating system to swap virtual memory to disk one page at a time. This would lead to a more graceful performance degradation if physical memory became a bottleneck.

The aforementioned changes are all implementation-specific, rather than to the overall approach I took, and I feel they are very likely to offer modest performance improvements. I would also like to consider some more radical design changes to the storage and partitioning layers. I continue to feel that peer-to-peer approaches are worthwhile, as they are self-configuring and easy for end users

to set up. However, I am not convinced that using hash-based partitioning for individual tuples is necessarily the best solution. I'm also not sure that it's not, but it would be interesting to explore the read, write, and query performance of alternate partitioning schemes.

One problem with the current reliable versioned data access layer is that the primary keys for each tuple are stored *three times* on disk. The tuple ID (containing the key and a version number) is store in the index page, and as the key in persistent storage; the primary key is then stored again as part of the complete tuple, also in persistent storage. For wide tuples with a small primary key (such as an integer), this is not necessarily an excessive amount of overhead. For narrower tuples, or tuples with, say, a string as a primary key, the overhead can be very high.

An incremental evolution is to store the entire tuple in the index page, and avoid the per-tuple persistent storage entirely. Persistent storage starts to look very MapReduce-inspired, as it becomes a file partitioned by hash key. This would negate the benefit of the initial motivation of index pages, which was to provide a level of indirection and thereby reduce the cost of making an update. It would be interesting to explore the benefits of this alternate approach experimentally. It would not be a drastic change, since it is a natural generalization of the existing covering index scan; recall that this is used for queries that only refer to attributes in the primary key.

An additional change to the way data is retrieved would be to support secondary indices. These would be arranged by the value of one or more attributes, instead of by hash key, and would correspond closely to unclustered indices in a traditional RDBMS. Given the high cost of using them (tuple IDs would have to be rehashed), they would only be useful for very selective predicates over the index attributes. This is also the case in a RDBMS, since a sequential scan is typically much faster than a large number of scattered reads. Detailed experiments would have to be performed to ensure that the query optimizer has a good understanding of when to use the secondary indices.

It would also be interesting to explore range-based partitioning, rather than hash-based partitioning. The global knowledge of the index structure could be used to partition the data using the primary key, rather than a hash of it, in a manner than somehow compensates for data skew. This avoids the (sometimes) expensive cost of computing a hash function, and if the primary key has semantic meaning, can be used to restrict the set of nodes that have participant in a scan for certain queries. It is not clear if such a scheme could be extended beyond base relations. In principle one could use the optimizer's histograms to compute an expected even partitioning by range for intermediate results; given how notoriously inaccurate cardinality (let alone distribution!) estimate are for intermediate results,

especially further from the leaves of a query plan, trusting them to perform data distribution could be somewhat risky. It is at least an idea worth considering.

Finally, it would be nice to implement background replication of data. We did not feel it necessary to do so, since the design would be very close to FreePastry's replication manager, and the experiments performed for this thesis did not require it. Still, there are interesting engineering challenges to making it work reliably and efficiently, and it is clear that it is a necessary feature for ORCHESTRA to be used in practice.

7.3 Load Balancing

There are several directions in which we would like to improve our approach to load balancing. One is simply to use other constraint solvers, including perhaps distributed, multi-threaded, or both, to produce higher quality partitionings more quickly without changing our fundamental approach. As the number of nodes grows, solving the routing problem to create an optimal (or close to optimal) partitioning becomes much more complex; it is reasonable to exploit the greater resources available to the system to improve its performance. We only explored relatively small system sizes. As it becomes more difficult to examine (even a small fraction) of the possible partitionings of the key space, it may be necessary to alter the objective function in some way to help guide the search; as the current objective function only considers the *most* imbalanced node, it does not necessarily force any amount of balancing between the other nodes, except than that they cannot be more imbalanced than the most imbalanced node. We hypothesize that this may make it difficult to compare the relative quality of two partitions. Even though they have the same routing imbalance, one may be generally much more balanced that the other. Even if this doesn't help guide the search to a solution with less routing imbalance, it may at least help it create partitionings that are more even aside from one (or a few) overloaded nodes; as discussed in Section 5.2, this may produce partitionings that perform better under transient spikes in load or network congestion.

A second direction is to remove low-powered or overly constrained nodes from the balancing problem presented to the constraint solver. If the optimizer cannot make good use of nodes (perhaps because they are so underpowered that assigning even a single routing chunk to them would be too expensive, or because they have so little of the key space available to them that their benefit is negligible), then even considering them makes the routing problem needlessly more complex.

A third direction is to consider routing balancing on a per-query basis, or at least to group queries into broad categories based on their dominant costs (such as network traffic, memory, CPU use, or disk) and to create a routing partitioning for each such category. This will reduce the benefit of caching, of course, but under certain circumstances it could be beneficial. A related optimization could increase the throughput of simultaneous queries. It may be faster to create a number of routing tables, each using only a subset of the nodes: if the per-query overhead at each node is high, then running fewer queries per node could increase query throughput. It could also be interesting to create different routing tables using different numbers of nodes; this could allow more nodes to be used for more complex or more important queries, and fewer for other queries. In approaches relying on differing routing tables, however, it could be difficult to consider the interactions between queries; this is, of course, also a problem in traditional single-server RDBMSs.

Finally, it could also be interesting to see the benefit of combining our routing optimization work with existing work on load balancing, in particular those described in Section 6.4 that do not use virtual servers. We expect that if the nodes are at least (somewhat) evenly loaded as a baseline, there is a higher chance that routing optimization will be able to achieve a good-quality result. Given the complexity of some of the load balancing approaches, however, this could be quite an undertaking. If changes are made to the reliable storage layer, as suggested in the previous section, it might be possible to consider approaches that make more extensive use of virtual servers as well; index page fragmentation would then no longer be an issue.

Chapter 8

Conclusions

In this dissertation, I have presented an end-to-end picture of the Orchestra collaborative data sharing system. I have described the need for collaborative data sharing, and explained why existing work did not meet the needs of loose, *ad hoc* sharing of structured data. I described the Orchestra model, where each participant has a local instance in their own schema, over which they make updates and pose queries. Orchestra is responsible for synchronizing this instance with the rest of the system, though complete agreement is not required or expected; a key focus of Orchestra is to allow incomplete agreement between participants in the system.

I reviewed the work done by others to enable update exchange between participants with different schemas, while maintaining a detailed provenance record of each update. I showed how we can add support for integrity constraints, transactional atomicity, and transactional dependency, to create more semantically meaningful data instances. I verified that we can do so efficiently.

The focus of this thesis, though, was on my work to enable truly distributed execution of Orchestra. The higher-level work described in Chapters 2 and 3 needs some sort of storage and query layer to execute over. In this thesis, I developed a peer-to-peer storage and query layer to meet the needs of Orchestra. It is easy to set up, and self-configuring; this is important, as it enables an Orchestra instance to be run by end users, instead of computer professionals. It is highly reliable, guaranteeing that data is never silently lost, and that queries return correct and complete answers. This is important because Orchestra is not tolerant of incorrect answers to its internal queries; such errors would likely cause incorrect user instances. I verified that the Orchestra storage and query layer performs well on a wide variety of queries, including a number of data sharing inspired-queries, and

that is scales well with both the number of nodes and the amount of data. This is important because neither may be known at the outset of an Orchestra-enabled collaboration.

The work presented here enables the full vision of Orchestra. The Orchestra model was designed to provide a low barrier to entry. Participants need to write a few simple mappings to one other participant in the system, and they can begin to automatically share data with the entire rest of the system. By making the implementation self-configuring, self-tuning, and able to run on a heterogeneous collection of commodity computers, the administrative barriers to entry become as low as the technical ones. Orchestra becomes a truly "plug and play" solution for data sharing.

Bibliography

Karl Aberer. P-Grid: A self-organizing access structure for P2P information systems. In *Cooperative Information Systems, 9th International Conference, CoopIS 2001, Trento, Italy, September 5-7, 2001*, 2001.

Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922--933, 2009.

Bogdan Alexe, Wang Chiew Tan, and Yannis Velegrakis. STBenchmark: towards a benchmark for mapping systems. *Proceedings of the VLDB Endowment*, 1(1):230--244, 2008.

Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, Cambridge, 2003.

Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the Seventeenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 31-June 2, 1999, Philadelphia, Pennsylvania, USA*, pages 68--79, 1999.

Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Transactions on Database Systems*, 33(1), 2008.

Philip A. Bernstein, James B. Rothnie Jr., Nathan Goodman, and Christos H. Papadimitriou. The concurrency control mechanism of SDD-1: A system for distributed databases (the fully redundant case). *IEEE Transactions on Software Engineering (TSE)*, 4(3):154--168, 1978.

Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie Jr. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems (TODS)*, 6(4):602--625, 1981.

Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47--76, 1987.

Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory --- ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*, pages 316--330. Springer, 2001.

Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang Chiew Tan. Archiving scientific data. In *SIGMOD 2002, Proceedings of the ACM SIGMOD International Conference on Management of Data, June 3-6, 2002, Madison, Wisconsin, USA*, pages 1--12, 2002.

Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Logical foundations of peer-to-peer data integration. In *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 241--251, 2004.

Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398--461, 2002.

Stefano Ceri, Maurice A. W. Houtsma, Arthur M. Keller, and Pierangela Samarati. Independent updates and incremental agreement in replicated databases. *Distributed and Parallel Databases*, 3 (3):225--246, 1995.

Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR 2003: First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003*, 2003.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehouse and OLAP technology. *SIGMOD Record*, 26(1), March 1997.

Surajit Chaudhuri and Vivek Narasayya. TPC-D data generation with skew. Technical report, Microsoft Research, 1999. Available via anonymous FTP from ftp.research.microsoft.com/users/viveknar/tpcdskew.

Brent N. Chun, Joseph M. Hellerstein, Ryan Huebsch, Shawn R. Jeffery, Boon Thau Loo, Sam Mardanbeigi, Timothy Roscoe, Sean C. Rhea, Scott Shenker, and Ion Stoica. Querying at Internet-scale. In *SIGMOD 2004, Proceedings of the ACM SIGMOD International Conference on Management of Data, June 13-18, 2004, Paris, France*, pages 935--936, 2004. Demonstration description.

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277--1288, 2008.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, second edition, 2001.

Yingwei Cui. *Lineage Tracing in Data Warehouses*. PhD thesis, Stanford University, 2001.

Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating System Principles, October 21-24, 2001, Chateau Lake Louise, Banff, Alberta, Canada (SOSP '01)*. ACM, 2001.

Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a dht for low latency and high throughput. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA*, pages 85--98, 2004.

Susan B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems (TODS)*, 9(3):456--481, 1984.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), December 6-8, 2004, San Francisco, California, USA*, pages 137--150, 2004.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM (CACM)*, 53(1):72--77, 2010.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007 (SOSP '07)*, pages 205--220, 2007.

Rina Dechter. *Constraint Processing*. Morgan Kaufmann, San Francisco, 2003.

David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commununications of the ACM (CACM)*, 35(6):85--98, 1992.

Peter Druschel and Antony Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of HotOS-VIII: 8th Workshop on Hot Topics in Operating Systems, May 20-23, 2001, Elmau/Oberbayern, Germany*, pages 75--80, Los Alamitos, CA, USA, 2001. IEEE Computer Society. ISBN 0-7695-1040-X.

W. Keith Edwards, Elizabeth D. Mynatt, Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, and Marvin M. Theimer. Designing and implementing asynchronous collaborative applications with Bayou. In *Proceedings of the 10th annual ACM Symposium on User Interface Software and Technology, Banff, Alberta, Canada*, pages 119--128, 1997.

Robert S. Epstein, Michael Stonebraker, and Eugene Wong. Distributed query processing in a relational data base system. In *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 31 - June 2, 1978*, pages 169--180, 1978.

Ronald Fagin, Phokion Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. *Theoretical Computer Science*, 336:89--124, 2005.

Ronald Fagin, Laura M. Haas, Mauricio A. Hernández, Renée J. Miller, Lucian Popa, and Yannis Velegrakis. Clio: Schema mapping creation and data exchange. In *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 198--236. Springer, 2009.

J. Nathan Foster. *Bidirectional Programming Languages*. PhD thesis, University of Pennsylvania, 2009.

J. Nathan Foster and Benjamin C. Pierce. *Boomerang Programmer's Manual*, 2008. Available from http://www.seas.upenn.edu/~harmony/.

J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences (JCSS)*, 73(4): 669--689, 2007.

Marc Friedman, Alon Y. Levy, and Todd D. Millstein. Navigational plans for data integration. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI 1999), July 18-22, 1999, Orlando, Florida, USA*, pages 67--73, 1999.

Prasanna Ganesan, Mayank Bawa, and Hector Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB 2004, Proceedings of 30th International Conference on Very Large Data Bases, August 29-September 3, 2004, Toronto, Canada*, pages 444--455, 2004.

Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey Ullman, and Jennifer Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2):117--132, March 1997.

Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy*, pages 98--102, 2006.

Shahram Ghandeharizadeh, Richard Hull, and Dean Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Transactions on Database Systems*, 21(3): 370--426, 1996.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003 (SOSP '03)*, pages 29--43, 2003.

Goetz Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD 1990, Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 102--111, New York, NY, USA, 1990. ACM. ISBN 0-89791-365-5.

Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 173--182. ACM Press, 1996.

Todd J. Green. *Foundations and Applications of Collaborative Data Sharing*. PhD thesis, University of Pennsylvania, 2009.

Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *VLDB 2007, Proceedings of 32nd International Conference on Very Large Data Bases, September 25-27, 2007, Vienna, Austria*, 2007a. Amended version available as Univ. of Pennsylvania report MS-CIS-07-26.

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, 2007b.

Todd J. Green, Nicholas Taylor, Grigoris Karvounarakis, Olivier Biton, Zachary Ives, and Val Tannen. Orchestra: Facilitating collaborative data sharing. In *SIGMOD 2007, Proceedings of the ACM International Conference on Management of Data, June 11-14, 2007, Beijing, China*, 2007c. Demonstration description.

Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. Efficient routing for peer-to-peer overlays. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA*, San Francisco, CA, March 2004.

Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 157--166. ACM Press, 1993.

Laura M. Haas, Renée J. Miller, Donald Kossmann, and Martin Hentschel. A first step towards integration independence. In *2nd International Workshop on New Trends in Information Integration (NTII 2010), 5-6 March 2010, Long Beach, California, USA*, 2010.

Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the Second USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2007, Boston, Massachusetts*, 2005.

Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema mediation in peer data management systems. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 505--516. IEEE Computer Society, March 2003.

Brad Hammond. WINGMAN: A replication service for Microsoft Access and Visual Basic. Technical report, Microsoft Corporation, 199? Available from `http://research.microsoft.com/en-us/um/people/gray/wingman_replicas.doc`.

Ryan Huebsch. *PIER: Internet scale P2P Query Processing with Distributed Hash Tables*. PhD thesis, University of California, Berkeley, May 2008.

Ryan Huebsch, Brent N. Chun, Joseph M. Hellerstein, Boon Thau Loo, Petros Maniatis, Timothy Roscoe, Scott Shenker, Ion Stoica, and Aydan R. Yumerefendi. The architecture of PIER: an Internet-scale query processor. In *CIDR 2005: Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005*, pages 28--43, 2005.

Jeong-Hyon Hwang, Magdalena Balazinska, Alex Rasin, Ugur Çetintemel, Michael Stonebraker, and Stanley B. Zdonik. High-availability algorithms for distributed stream processing. In *Proceedings of the 21st International Conference on Data Engineering, April 5-8, 2005, Tokyo, Japan*, pages 779--790, 2005.

Jeong-Hyon Hwang, Ying Xing, Ugur Çetintemel, and Stanley B. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *Proceedings of the 23rd International Conference on Data Engineering*, pages 176--185, 2007.

Zachary Ives, Nitin Khandelwal, Aneesh Kapur, and Murat Cakir. ORCHESTRA: Rapid, collaborative sharing of dynamic data. In *CIDR 2005: Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005*, pages 107--118, January 2005.

Zachary G. Ives, Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Val Tannen, Partha Pratim Talukdar, Marie Jacob, and Fernando Pereira. The ORCHESTRA collaborative data sharing system. *SIGMOD Record*, 2008.

James B. Rothnie Jr., Philip A. Bernstein, Stephen Fox, Nathan Goodman, Michael Hammer, Terry A. Landers, Christopher L. Reeve, David W. Shipman, and Eugene Wong. Introduction to a system for distributed databases (SDD-1). *ACM Transactions on Database Systems (TODS)*, 5(1):1--17, 1980.

David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Peer-to-Peer Systems III, Third International Workshop, IPTPS 2004, La Jolla, CA, USA, February 26-27, 2004, Revised Selected Papers*, pages 131--140, 2004.

David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997 (STOC '97)*, pages 654--663, 1997.

Grigoris Karvounarakis. *Provenance in Collaborative Data Sharing*. PhD thesis, University of Pennsylvania, 2009.

Grigoris Karvounarakis and Zachary G. Ives. Bidirectional mappings for data and update exchange. In *11th International Workshop on the Web and Databases, WebDB 2008, Vancouver, BC, Canada, June 13, 2008*, 2008.

Anastasios Kementsietsidis, Marcelo Arenas, and Renée J. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *SIGMOD 2003, Proceedings of the ACM SIGMOD International Conference on Management of Data, June 9-12, 2003, San Diego, California, USA*. ACM, June 2003.

Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the 20th annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), August 26-29, 2001, Newport, Rhode Island, USA*, August 2001.

YongChul Kwan, Magdalena Balazinska, and Albert Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. In *VLDB 2008, Proceedings of 33nd International Conference on Very Large Data Bases, August 26-28, 2008, Auckland, New Zealand*, pages 574--585, 2008.

Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133--169, 1998.

Jonathan Ledlie and Margo I. Seltzer. Distributed, secure load balancing with skew, heterogeneity and churn. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies, 13-17 March 2005, Miami, FL, USA*, pages 1419--1430, 2005.

Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Source inconsistency and incompleteness in data integration. In *Proceedings of the 9th International Workshop on Knowledge Representation meets Databases (KRDB 2002), Toulouse France, April 21, 2002*, April 2002.

Maurizio Lenzerini. Tutorial - data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 3-5, 2002, Madison, Wisconsin USA*, 2002.

Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 251--262. Morgan Kaufman, 1996.

Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, and Robert A. Yost. Computation and communication in R*: a distributed database manager. *ACM Transactions on Computer Systems*, 2 (1):24--38, 1984. ISSN 0734-2071.

Boon Thau Loo, Joseph M. Hellerstein, Ryan Huebsch, Scott Shenker, and Ion Stoica. Enhancing P2P file-sharing with an Internet-scale query processor. In *VLDB 2004, Proceedings of 30th International Conference on Very Large Data Bases, August 29-September 3, 2004, Toronto, Canada*, pages 432--443, 2004.

Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB'86, Proceedings of 12th International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan*, pages 149--159, 1986.

Gurmeet Singh Manku. Balanced binary trees for ID management and load balance in distributed hash tables. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 197--205, 2004.

C. Mohan, Bruce G. Lindsay, and Ron Obermarck. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4):378--396, 1986.

Richard Mortier, Dushyanth Narayanan, Austin Donnelly, and Antony I. T. Rowstron. Seaweed: Distributed scalable ad hoc querying. In *Proceedings of the 22nd International Conference on Data Engineering Workshops, ICDE 2006, 3-7 April 2006, Atlanta, GA, USA (NetDB '06)*, page 30, 2006.

Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), December 9-11, 2002, Boston, Massachusetts, USA*, 2002.

Dushyanth Narayanan, Austin Donnelly, Richard Mortier, and Antony I. T. Rowstron. Delay aware querying with Seaweed. *VLDB Journal*, 17(2):315--331, 2008.

Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD 2008, Proceedings of the ACM International Conference on Management of Data, June 10-12, 2007, Vancouver, Canada*, pages 1099--1110, 2008.

Gultekin Özsoyoglu and Richard T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. Knowl. Data Eng.*, 7(4):513--532, 1995.

D. Storr Parker, Jr, Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David A. Edwards, Stephen Kiser, and Charles S. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Software Eng.*, 9(3):240--247, 1983.

Karin Petersen, Mike Spreitzer, Douglas B. Terry, Marvin Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, October 5-8, 1997, St. Malo, France (SOSP '97)*, pages 288--301, 1997.

Benjamin C. Pierce and Jérôme Vouillon. What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.

C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, Sebastopol, CA, second edition edition, 2008.

Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulliten*, 23(4):3--13, 2000.

Louiqa Raschid and Stanley Y. W. Su. A parallel processing strategy for evaluating recursive queries. In *VLDB'86, Proceedings of 12th International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan*, pages 412--419, 1986.

Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM 2001, August 27 - 31, San Diego, CA USA*, 2001.

Peter L. Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Summer 1994 Technical Conference, June 6-10, 1994, Boston, Massachusetts*, pages 183--195, 1994.

Michael K. Reiter. A secure group membership protocol. *IEEE Trans. Software Eng.*, 22(1):31--42, 1996.

Sean C. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *USENIX Annual Technical Conference, General Track*, pages 127--140, 2004.

Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of IFIP Int. Conf. on Distributed Systems Platforms (Middleware)*, pages 329--350, November 2001.

Fereidoon Sadri. Aggregate operations in the information source tracking method. *Theor. Comput. Sci.*, 133(2):421--442, 1994.

Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42--81, 2005.

Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput.*, 39(4):447--459, 1990.

Mehul A. Shah. *Flux: A Mechanism for Building Robust, Scalable Dataflows*. PhD thesis, University of California, Berkeley, 2004.

Jim Smith and Paul Watson. Fault-tolerance in distributed query processing. In *International Database Engineering and Applications Symposium*, pages 329--338, 2005.

Jim Smith, Paul Watson, Sandra de F. Mendes Sampaio, and Norman W. Paton. Polar: An architecture for a parallel ODMG compliant object database. In *Conference on Information and Knowledge Management*, pages 352--359, 2000.

Jim Smith, Anastasios Gounaris, Paul Watson, Norman W. Paton, Alvaro A. A. Fernandes, and Rizos Sakellariou. Distributed query processing on the grid. In *GRID*, pages 279--290, 2002.

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM 2001, August 27 - 31, San Diego, CA USA*, 2001.

Michael Stonebraker. The design of the POSTGRES storage system. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 289--300. Morgan Kaufmann, 1987. ISBN 0-934613-46-X.

Sonesh Surana, Brighten Godfrey, Karthik Lakshminarayanan, Richard M. Karp, and Ion Stoica. Load balancing in dynamic structured peer-to-peer systems. *Performance Evaluation*, 63(3):217--240, 2006.

Tandem Database Group. NonStop SQL, a distributed, high-performance, high-availability implementation of SQL. Technical report, HP Labs, April 1987. Report TR-87.4.

Nicholas E. Taylor and Zachary G. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD 2006, Proceedings of the ACM International Conference on Management of Data, June 27-29, 2006, Chicago, IL*, pages 13--24. ACM, 2006.

Nicholas E. Taylor and Zachary G. Ives. Reliable storage and querying for collaborative data sharing systems. In *Proceedings of the 26th International Conference on Data Engineering, March 1-6, 2010, Long Beach, CA*, pages 40--51, 2010.

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, September 28-30, 1994*, pages 140--149, 1994.

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive --- a petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th International Conference on Data Engineering, March 1-6, 2010, Long Beach, CA*, pages 996--1005, 2010.

Edward Tsang. *Foundations of constraint satisfaction*. Academic Press, London, 1993.

Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I: Classical Database Systems. Computer Science Press, Rockville, MD, 1988.

Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR 2005: Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005*, 2005.