Look-up Tables: The Benefit of Enabling Fine-Grained Routing and Load Balancing

by

Aubrey Lynn Tatarowicz

S.B., E.E.C.S., Massachusetts Institute of Technology, 2010

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

June 2011

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2011

Certified by .....
Samuel R. Madden
Associate Professor
Thesis Supervisor
May 20, 2011

Accepted by .....
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Look-up Tables: The Benefit of Enabling Fine-Grained Routing and Load Balancing

by

Aubrey Lynn Tatarowicz

Submitted to the Department of Electrical Engineering and Computer Science
May 20, 2011
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Data volumes are exploding. It is essential to use multiple machines to store such large amounts of data. To address this explosion, storage systems like databases need to be distributed across many machines.

Transactions that access a few tuples, often seen in web workloads such as Twitter, do not run optimally using traditional partitioning schemes [25]. Hence, increasing the number of machines often presents a bottleneck for workloads where each transaction accesses just a few tuples.

Fine-grained partitioning can fix the scale out problem introduced by simplistic partitioning schemes. In this thesis, I introduce a design of a distributed query execution system that handles fine-grained partitioning using look-up tables. I introduce *look-up tables*, which is a mapping from a tuple attribute to a tuple back-end location such that fine grained partitioning can be supported.

I show through both synthetic and real data that fine-grained partitioning enabled by look-up tables can increase throughput of a distributed database system. My goal is scale-out with the number of machines used in the distributed database. I show in my experiments that scale-out can be reached if an ideal partitioning can be created. I test my implementation on a Wikipedia data set. I show in this example a factor of three times better performance compared to the optimal hash partitioning scheme with eight back-ends and signs of continual scale-out with more machines. Through the use of large data sets and projecting my results onto even larger data sets, I show that look-up tables can be used to represent complex partitioning schemes for databases containing billions of tuples.

Thesis Supervisor: Samuel R. Madden
Title: Associate Professor

Acknowledgments

I am thankful for those people that helped me and supported me in the endeavor to complete my thesis.

I would like to show my gratitude to my supervisor, Sam Madden, whose encouragement, guidance, and support helped me mold this thesis into something substantial.

This thesis would not have been possible without the help of my fellow grad (and post doc) students Evan Jones and Carlo Curino. Thank you for white boarding ideas with me and helping me hone in on my design. Without the base code that they spent many hours writing, this thesis would not have been possible. Thanks to Carlo for providing his wisdom and help in partitioning the data which my thesis so very much depended on. Evan, thanks for debugging with me as we tried to figure out the problems in these multi-threaded Java programs.

I would also like to thank my family and friends for their support in my long work hours to finally graduate. Thanks Mom and Dad for encouraging me throughout my M.I.T. career. A special thanks goes to my boyfriend, Dember, whose love and support got me through writing my thesis and just about everything else here at M.I.T. You have the unique ability to calm me down when I am stressed.

Finally, I would like to thank everyone who directly or indirectly helped me with this thesis, especially everyone in the DB group.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data volumes are exploding. It is estimated that Twitter has around 200 million users [23]. Facebook has approximately 300 million users and more that 3.9 trillion feed actions processed per day [27]. It is essential to use multiple machines to store such large amounts of data. For example, Facebook is running approximately 30,000 servers to hold and process data [27]. To address this explosion, storage systems like databases need to be distributed across many machines.

The primary method in which databases are scaled is through horizontal partitioning, where different rows of table are placed on different machines. By placing data on separate nodes, it is possible to achieve near linear speedup when executing analytical queries. Since these queries scan large amounts of data, which can be accessed in parallel in a distributed database. Transactions that access a few tuples, often seen in web workloads such as Twitter, do not run optimally using traditional horizontal partitioning schemes [25]. The problem is that if a small number of tuples are accessed on several machines, there is no benefit of parallel I/O (input/output), but the query is distributed which adds network I/O and substantially slows execution.

Hence, increasing the number of machines often presents a bottleneck for workloads where each transaction accesses just a few tuples. Instead of partitioning data randomly (e.g. using hashing) or via fixed partitioning of ranges, what is needed is a fine-grained partitioning scheme that places tuples that are accessed together in the same partition.

Fine-grained partitioning can fix the scale out problem introduced by simplistic partitioning schemes. In this thesis, I introduce a design of a distributed query execution system

that handles fine-grained partitioning using look-up tables. I introduce *look-up tables*, which is a mapping from a tuple attribute to a tuple back-end location. Look-up tables map the location of a tuple by storing a column value and back-end location for the tuple. They enable fine-grained partitioning where hash or round robin limit the partitioning scheme to something arbitrary. A basic look-up table is created for one column of a table, mapping the value of a column to the tuple location on the back-ends. This column is normally a unique id for the tuple. The distributed query execution engine uses the look-up table to find tuple locations for a query so that it can route the query to the correct back-ends.

In Chapter 2, I discuss the related work in the field of distributed databases. I explain how these related works do not solve the problem of transactions accessing a few tuples over multiple back-ends. In Chapter 3, I describe a broad picture of the system and how it fits into the Relational Cloud project [9] in which my thesis is a component.

In Chapter 4, I delve into the design details and how I minimize network I/O and use the information stored in look-up tables to make smart routing decisions to the distributed database back-ends. I also discuss how the router handles hundreds of transactions simultaneously. In Chapter 5, I go into more detail on the implementation in Java.

In Chapter 6, I show through both synthetic and real data that fine-grained partitioning enabled by look-up tables can increase throughput of a distributed database system. Furthermore, I show that when using an ideal partitioning, we see near-linear scale-up with the number of machines even when traditional partitioning schemes achieve little or no scale-up. I also test my implementation on actual Wikipedia data obtained from Mediawiki, the company behind Wikipedia. I show in this example a factor of three times better performance compared to the optimal hash partitioning scheme with eight back-ends. I also show in this example that my system shows signs of continual scale out with more partitions where the hash partitioning scheme has a performance ceiling.

Look-up tables must be stored in memory, where as hashing or round robin are functions of the tuple value and only need a quick operator to compute the back-end. In Chapter 7, I address the scalability of look-up tables. A key concern of look-up tables is the ability for look-up tables to scale to large data sets. I show through the use of large data sets and projecting my results onto even larger data sets that look-up tables can be used to represent complex partitioning schemes for databases containing billions of tuples.

In Chapter 8, I discuss the additional elements that would need to be created to make a

complete distributed execution system using look-up tables. Finally, in Chapter 9, I wrap up my thesis and make final conclusions.

Chapter 2

Related Work

In this chapter, I discuss related work and the problems that these works address. I explain that each has a trade-off and does not solve the difficulty of scaling these large databases with transactional web workloads. In this chapter, you will gather a better understanding of how my thesis research fits into the world of databases and the problem it solves. You will also learn that existing research has not solved the scale-out goal with web workloads. The goal is to scale throughput with the number of machines in the distributed database.

2.1 Parallel Databases

A fair amount of research has gone into distributed and parallel databases to handle OLAP workloads such as R* [15]. Online Analytical Processing (OLAP) refers to longer running processes that often compute statistics over large amounts of data. OLAP transactions are often run over historical data and are expected to take a long time to run as statistics are computed.

These traditional databases enforce Atomicity, Consistency, Isolation, and Durability (ACID) guarantees. ACID guarantees are often seen in computer systems in general and are necessary in transactional systems such as banking applications.

Traditionally, parallel databases follow the design of a shared nothing architecture to distribute the work among nodes. In a shared nothing database system, each node stores a fraction of the data, and that data is only accessed by that node. The way information is transferred between CPUs via an interconnection network [21]. See Figure 2-1 on the general design of a shared nothing architecture.

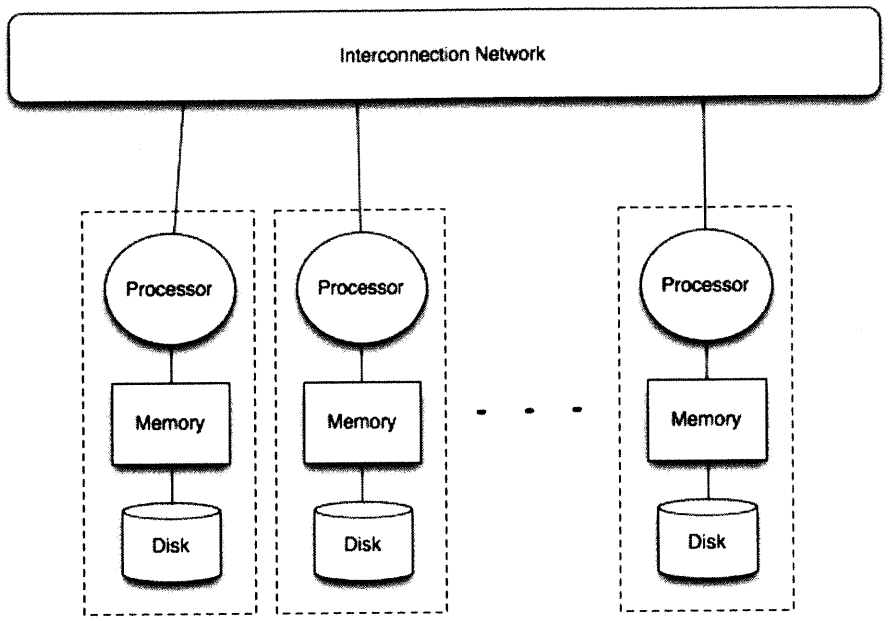

Figure 2-1: Shared Nothing Architecture.

In contrast to parallel databases aimed at optimizing for OLAP workloads, in this thesis, I focus on web or Online Transaction Processing (OLTP) workloads. OLTP refers to short online transactions such as individual updates, inserts, or deletes. OLTP workloads emphasize quick response to user input. These traditional horizontal partitioned parallel databases do not work well with transactions that access a few tuples, often seen in these web workloads. The problem is that if a small number of tuples are accessed on several machines, there is no benefit of parallel I/O (input/output), but the query is distributed which adds network I/O and substantially slows execution.

2.2 Cloud Computing

Cloud computing systems are based on a "pay-as-you-go" usage model where users can demand resources when needed and release resources as desired, paying for only what they use. Many cloud systems are accessible over the Internet so that people can acquire and release resources online. For example, Amazon's cloud services [1] are popular among many companies including Dropbox [28], Quora, Reddit [22], NASA Jet Propulsion Lab, Washington Post, Yelp, and PBS [4]. The draw of cloud services is the simplicity of setup and low up front cost as compared to setting up a privately owned network of computing

power. Several different types of clouds exist: infrastructure clouds, which provide storage or computer resources; platform clouds (e.g. SQL Azure [24]) which provide support for running applications in the cloud; and application creation clouds (e.g. Google Docs [12]) [2, 3].

The research presented in this thesis, a distributed query execution system, is a component of Relational Cloud, a database-as-a-service cloud.

2.3 Relational Cloud

Relational Cloud is a project that is aimed providing a "pay-as-you-go" database cloud that can host a large and diverse set of OLTP/web applications, providing transactional guarantees, scalability, and efficient resource utilization.

The intention of this project is to bring the power and versatility of cloud computing to databases. Creating and maintaining a database can be cumbersome for small organizations. For large organizations, having many small databases can be a waste of resources. Hence, two goals of this project are to consolidate data management for large organizations and outsource data management to a cloud-based service provider for smaller organizations [9].

Relational Cloud partitions clients' data across many machines. The client does not know how information is partitioned among the database, instead interacting with Relational Cloud as it would with a centralized database management system [9].

In addition, Relational Cloud is workload aware so that data can be co-located on partitions in order to reduce the overhead of distributed queries and transactions by locating data that is accessed together on the same physical node, which, as described above, is important for OLTP/web workloads. For OLAP workloads data may distributed in order to push down joins and parallelize the work over all nodes in order to process queries/transactions quicker.

This thesis describes a component of Relational Cloud. The research in this project allows Relational Cloud to leverage partitioning strategies such as Schism [10] in creating fine-grained partitioning in order to optimize for particular workloads.

2.4 Partitioning Approaches

Horizontal scaling grows the number of machines rather than the size of any particular machine as the workload or data set grows. Horizontal scaling in the ideal case sees linear speedup with the number of machines and directly follows the shared nothing design. The rest of this section describes different partitioning techniques developed to allow for horizontal partitioning.

2.4.1 Hash Partitioning / Round Robin Partitioning

In databases, the question remains as to where should data be located when the database scales horizontally. Common schemes used include hash and round robin partitioning. Hash partitioning applies some hash function to the data to place the data evenly among the back-ends. Round robin does the same thing by sending each successive tuple to a different partition. This works well on OLAP workloads (analytical queries that scan large amounts of data) where the time to answer a query depends mainly on disk access speed. Hash partitioning and round robin partitioning fail to be the best scheme for OLTP/web workloads [25].

2.4.2 Range Partitioning

Range partitioning is a partitioning technique where ranges of data are stored on different back-ends. Cut-points are defined on the edges of where one partition ends and another begins. Similar to hashing, range partitioning places data on arbitrary locations which may not be favorable for web workloads.

2.4.3 Schism

Schism introduces more complex partitioning strategies. It uses information on data access patterns to devise the optimal partitioning for the data. Schism shows through a series of experiments that distributed transactions are expensive in OLTP settings. Schism produces balanced partitions by analyzing the expected workload and producing a graph where edges connect tuples that are in transactions together. It then attempts to minimize distributed transactions by minimizing the number of cut edges while producing balanced partitions. Schism works well for n-to-n relationships, typical of social network databases. Schism's

output can (and is) used as the input to my distributed query execution system. [10].

2.4.4 Handling Online Social Networks

This section outlines current designs aimed at handling large social network data sets and the complicated workload on these sets.

The Little Engines that Could: Scaling Online Social Networks

Pujol et. al. describe a design, Social Partitioning and Replication (SPAR) middle-ware, that guarantees that one-hop neighboring data is co-located. They do this by partitioning the data in a smart way and then replicating one-hop neighbors on the same node as the "master" [20]. In the example of Facebook, all of a user's friends will be replicated on the node with that user [19].

In choosing this design, they achieve data locality for the trade-off of replicating data. For many partitions in complex graphs this may mean a significant amount of replication, which in turn increases the cost of updates. The system does not inherently support consistency. They mention that they may use something like Dynamo [11].

Facebook's Architecture

Facebook developed and eventually made open source their distributed database, Cassandra. Cassandra combines the ideas of Dynamo [11] and BigTable [7]. Cassandra is designed to scale to a very large size across many servers. Cassandra supports eventual consistency as a key-value store [14].

Twitter's Architecture

Twitter uses MemCache to make look-ups fast. Since requests to Tweets are often requested based on a time filter, Twitter partitions by time on Tweets rather than user id or tweet id. In order to easily look-up follows relations, Twitter replicates the Follows table to go in the opposite direction. Therefore, it is quick to look-up the relation in either direction. This table is partitioned by user id [13].

2.5 Key-Value Stores

Key-value stores have emerged in web applications as the current solution to the scaling difficulties of large databases with many concurrent reads and writes. Yahoo's PNUTS [8], Google's Bigtable [7], MongoDB [17], and MemcacheDB [16] are a few implementations of such key-value stores. Key-value stores limit the types of queries that can be run on a database fetching single rows or updating single rows by a key. They do not support joins or aggregates. Such functionality must be supported at the user-level. In doing so, there is no transactional guarantee for joins or queries more complex than single row operations.

Many key-value stores, including Bigtable [7] are eventually consistent, meaning that it is possible to read an old value of a tuple rather than the current value. Key-value stores relax ACID guarantees that traditional databases have. The aim of my thesis is to research an alternative solution to scale to web workload sizes without sacrificing the ACID guarantees the way key-value stores do. ACID is important in delivering reliable and consistent data to the user. In addition, my system will support useful functionality that traditional databases have such as joins which key-value stores leave out of the system.

Chapter 3

System Overview

In this chapter, I outline the overall system design for my distributed query execution system and how this design fits into the Relational Cloud project.

3.1 Relational Cloud Overall Design

Relational Cloud is a cloud database architecture. It follows the general design of a shared nothing architecture. Each database partition is stored on a single machine that is capable of executing query statements.

Both databases and tables may be split up across nodes. The basic architecture of Relational Cloud can be summarized in Figure 3-1.

The distributed transaction system sends a query to the router. The router must determine which partitions to access, and construct a query for each partition. For example, consider the following query.

```
SELECT * FROM mytable WHERE id IN(1,7)
```

Suppose `id = 1` is on partition 1 and `id = 7` is on partition 5. Then the query will be modified such that the following query is sent to partition 1:

```
SELECT * FROM mytable WHERE id IN(1)
```

The following query is sent to partition 5:

```
SELECT * FROM mytable WHERE id IN(7)
```

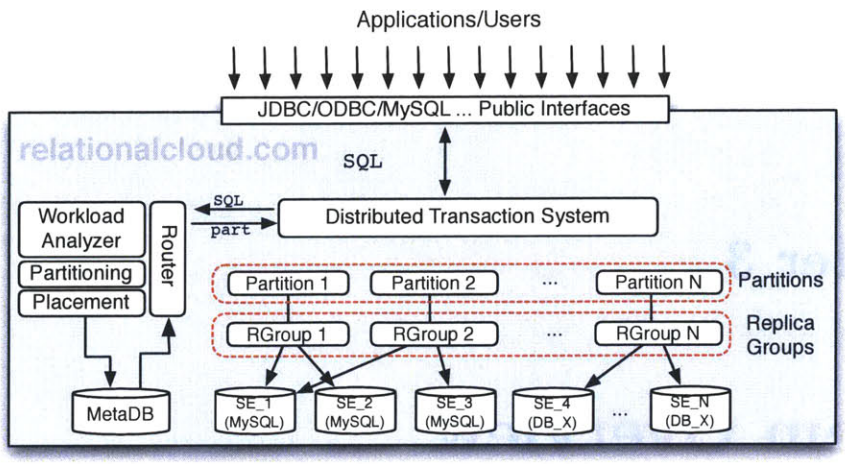

Figure 3-1: Relational Cloud Architecture [9].

The router also figures out how to combine separate results from different nodes. In this example, the results will be merged. The end result should be what would have been the result of the original running query on a centralized database. The focus of my thesis is the implementation of the router when fine-grained partitioning is in use. In particular, my router implementation stores this partition information in look-up tables.

3.2 Overall Router Design

The router and back-ends have been simplified from the design specified in Figure 3-1 in order to create a prototype router. The main changes in design is the removal of replica groups, the MetaDB, and Workload Analyzer. Additionally, the router communicates directly with MySQL back-ends rather than with nodes and replica groups as in Figure 3-1. Currently, clients connect directly to the router via a JDBC interface. Figure 3-2 shows the simplified router design.

3.3 Look-up Table Logic

A look-up table holds information about the partitions on which a set of tuples are stored. In the table, each tuple is identified by the value of some key attribute(s), and partitions are represented by an integer partition id. Figure 3-3 shows a simplistic database. Each back-end holds one partition of the table. The partitions have an id column, which is a primary key, along with a value. Table 3.3 shows the look-up table that is generated when id is used

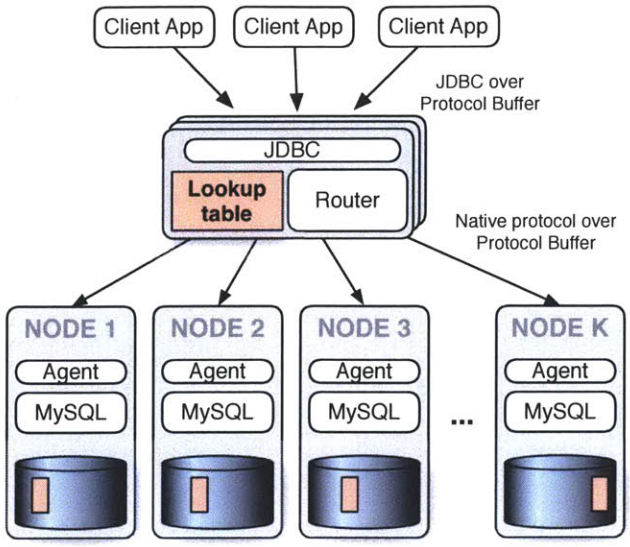

Figure 3-2: Router Architecture Overview.

as the look-up table key. The mapping goes from tuple id to partition number. There can be multiple underlying implementations of look-up tables, although one important feature is that look-up of a partition given a key should be quick and require constant time.

Given this overview of look-up tables, in the next chapter, I will describe the design of the router in more detail.

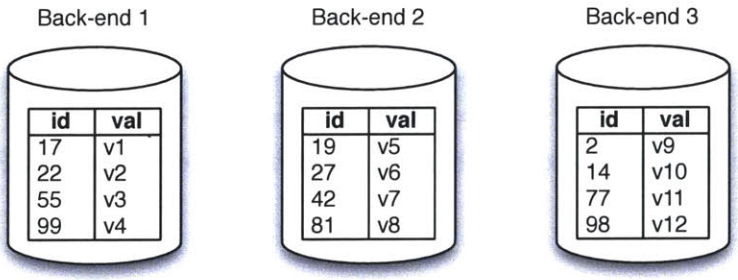

Figure 3-3: Example distributed database with data.

id	Back-end node id
2	3
14	3
17	1
19	2
22	1
27	2
42	2
55	1
77	3
81	2
98	3
99	1

Table 3.1: Example look-up table generated from the database in Figure 3-3

Chapter 4

Router Design

In this chapter, I discuss the router design. Figure 4-1 shows the architecture of the router. The router is designed to operate in parallel. The RouterServer that serves incoming requests assigns the work to a TransactionWorker that computes the partitions that participate in a query and generates the modified queries that those partitions execute. There can be several TransactionWorkers running at a time. The ConnectionWorkers handle the sending of the query to the back-end node and send the results back to the TransactionWorker once receipt of the results. The reason for ConnectionWorker is to have the distributed query sent to each node simultaneously.

The following sections describe the logic the router has in order to modify queries and how it chooses the correct back-ends in which to send the queries.

4.1 Query Manipulation

Since the database has multiple tables distributed over multiple back-ends, an incoming query from a client may need to be modified before being sent to a back-end to get correct results. The router needs to perform this transformation in addition to determining which nodes should participate in a query. In the rest of this section, I describe how the router works through a series of examples.

4.1.1 Filtering Queries

An `IN` query is a common query in applications such as social networks were information about a group of people or things needs to be acquired at once. It is also building block in

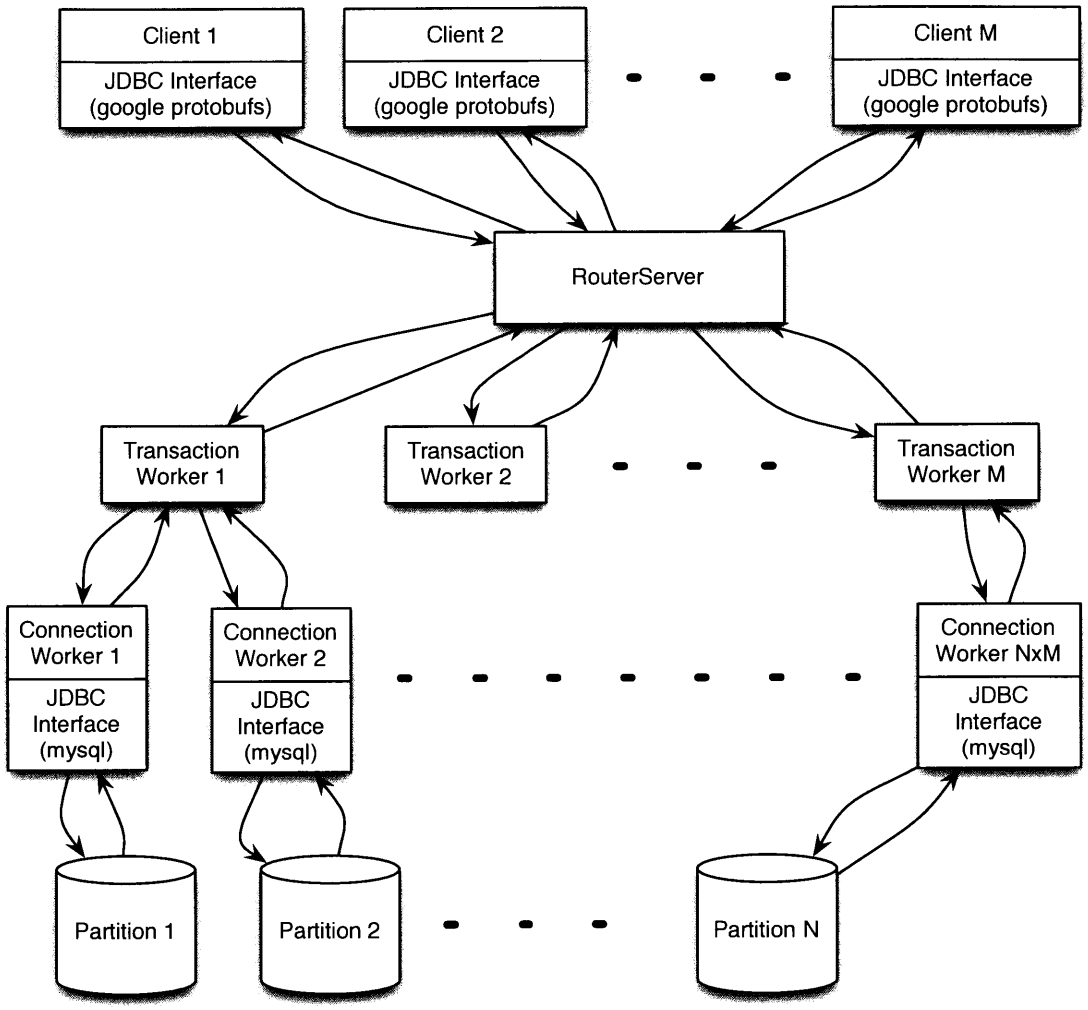

Figure 4-1: Router communication with clients and back-end nodes.

creating more complicated joins queries in rounds discussed in Section 4.1.2.

For this example, the simplistic distributed database in Figure 3-3 and the look-up table generated from the database in Table 3.3 is used.

Consider the following query sent to a router with the database described above:

```
SELECT * FROM mytable WHERE id IN (2, 19, 27, 77);
```

Suppose ids 2 and 77 are located on back-end 3 and ids 19 and 27 are located on partition 2. In this case, the following rewritten queries will be sent to the back-ends:

```
Back-end 2: SELECT * FROM mytable WHERE id IN (19, 27);
Back-end 3: SELECT * FROM mytable WHERE id IN (2, 77);
```

Rewriting the queries in this way ensures that each back-end only does the work necessary to find and return the correct data. In the case of this example, no queries are sent to back-end 1.

`IN` queries are not the only example of `SELECT` queries that are rewritten. The router currently is able to rewrite `SELECT` queries that include greater-than, greater-than-equals, less-than, less-than-equals, and equals operators. In the case of equals and `IN` queries, the number of back-ends necessary to which queries must be sent can be much fewer than the total number of partitions. Additionally, the router may be able to determine if the result set will be empty enabling the router to send the result to the user immediately.

4.1.2 Joins

Joins over multiple tables are an important function in many database applications.

The router has multiple choices for how to handle joins depending on the layout of the data and the information provided in the look-up tables. Several plans are described below, outlining their strengths and weaknesses.

Push-down Joins

A push-down join happens when the join is sent directly to the back-end node. This means that each back-end performs the join and the results are then combined by the router. Push-down joins are most likely the fastest and simplest model of distributed joins. A join can only be pushed down if it is known that data on one back-end does not need to be

joined with data from another back-end. This is the case when the join condition is the same as the partitioning condition. Push-down joins cannot be used when information from different back-ends must be joined together.

Optimistic Outer Joins

Optimistic outer joins are similar to push-down joins in that the back-end node does the work of joining the data. These joins apply to primary key - foreign key joins, where at most one primary key joins with any given foreign key. Typically, such queries includes a *bound variable* where either the primary or the foreign key value is assigned to some constant value in the query (otherwise these queries would involve cross products).

If the partitioning algorithm is good, in most cases such joins will be local to one back-end. Hence, the join is sent to the back-end indicated by the bound values in the query. Since some data required to answer the query may not be on that node, an outer join is performed. If a primary key matching a foreign key is found, the data returned is a full tuple in the set of results. In the case that the primary key is not found at the location, then the outer join results in `NULL` data where the joining attributes should be. If `NULL`s are present in the returned data set, the router sends an `IN` query to the back-ends with the tuples that have the needed values.

An example case to use an optimistic outer join is Twitter. Consider the following query:

```
SELECT * FROM follows f, user u ON f.f2 = u.uid WHERE f.f1 = 24;
```

Twitter has a `follows` table which is a relation in which user id `f1` follows user id `f2` where `f1` and `f2` are foreign keys referring to the users table primary key, `uid`. The join above selects information about the people who user id 24 follows. In an ideal partitioning, all of the data for a user's followers will be on the same back-end as that user. However, Twitter represents a complicated interconnection of users, so it may be that not all of the people a person follows are on the same back-end as that user. However, since there is a high probability that the query can be answered on the back-end with data on user 24, then the following query is sent to the back-end to which the look-up table (built on `followers.f1`) points for user 24.

```
SELECT * FROM follows f LEFT OUTER JOIN user u ON f.f2 = u.uid WHERE f.f1 = 24;
```

This join result will contain NULLs where `f.f2 = u.uid` did not find a matching `uid` on this back-end. For such results, an `IN` query will be sent requesting all of the data for the missing ids. In the case that there are no `NULL` values, then the query result is returned directly (requiring the router to only contact one back-end).

Joins in Rounds

Joins in rounds is useful when predicates for one of the tables is fairly selective. In this scenario, a `SELECT` over one table is performed, filtering on the predicates for that table. Then another `SELECT` is performed by sending a `IN` query on the returned results from the first round. This can be extended to support an arbitrary number of rounds and hence an arbitrary number of tables being joined.

Joins at the Router Level

The final and often slowest join plan is joining at the router level. This can be useful if all predicates on the tables are selective, and is a fall-back that will always work for joins. In this plan, each table is selected, retrieving all data from that table that satisfies predicates specific to the table. The join is performed by joining at the router.

Choosing the Correct Join Plan

In order to pick the correct join plan, statistics about the cardinality of the tables and information about location of tuples is necessary. This information can be stored at the router in order to make a smart distributed query plan. The focus of my thesis is to argue that look-up tables and fine-grained partitioning can solve the problem of large amounts of data that must be partitioned over many nodes. For this reason, I did not implement a query planner that chooses the correct join plan based on statistics and look-up table knowledge. Instead, this decision is currently hard-coded into the client. In the experiments section, the correct join plan is relayed from the client to match what a smart router would have chosen.

Chapter 5

Implementation Details

In this chapter, I discuss the important components in the implementation of the distributed execution system router. The router is built in Java.

5.1 Component Interaction

The RouterServer is the component that the clients talk to in order to run queries on the back-end databases. To the clients, it appears as if they are connecting to a single database with ACID guarantees. The clients use a JDBC interface in order to provide the normal interaction with a database for a Java program. The data is serialized using Protocol Buffers (Protobufs) [18] as a method of encoding the structure data for communication over the network.

Figure 4-1 on page 28 shows the communication model. Arrows denote the flow of information between components. There is one TransactionWorker per Client that connects to the RouterServer. A TransactionWorker receives a request that the client has made. The TransactionWorker generates queries each back-end needs to execute. To allow queries to be sent to multiple back-ends at once, each TransactionWorker is configured with one ConnectionWorker object per back-end. ConnectionWorkers communicate with their back-end, sending queries, and waiting for results.

The ConnectionWorker sends the results to the TransactionWorker, which completes any final work on the query, such as merging the results from multiple ConnectionWorkers. The TransactionWorker sends the completed results to the RouterServer to send back to the client.

The RouterServer and all components of the router are written in Java. The RouterServer and the components discussed in this chapter consists of approximately 4,000 lines of code (1,700 semicolons). This does not include the rest of the Relational Cloud code, but only the code specific for routing and look-up tables.

5.2 Key Components' Details

RouterServer

The RouterServer receives work and sends the work to a TransactionWorker. The underlying implementation uses a java.util.concurrent.ThreadPoolExecutor as a thread pool to carry out the TransactionWorker work. The RouterServer adds work to the thread pool by creating a Runnable TransactionWorker object with the query to be executed and the transaction id to execute on.

The RouterServer holds a central mapping of transaction id to the state of the transaction which is used by other components in order to keep state for transactions.

TransactionWorker

The TransactionWorker class implements Runnable. One instance is created for each client request. The TransactionWorker receives the sent query to execute, the transaction id, and if the transaction is auto-commit. The TransactionWorker uses the look-up table in order to determine the modified queries and the locations to which those queries should be sent.

In order for the system to be transactional, the TransactionWorker must know the state of the query, which includes the following information:

- If the client connection is auto-commit.
- The ConnectionWorker objects.
- The partitions that have had queries run on them.
- The state of the operation (idle, active, finishing, done).

Auto-commit information is needed for 2 major reasons. 1) If the query is auto-commit, then the query must be committed before returning results to the user and 2) If the query is

auto-commit and it only needs to run on one back-end, then the connection with that back-end can also be set to auto-commit. Each TransactionWorker needs a ConnectionWorker to each back-end that it communicates with in order to run queries on the back-end. In addition, for a transaction, the same connection to a back-end node must be used for all queries run in a transaction. For this reason, the TransactionWorker needs to ensure that it uses the same ConnectionWorker for a given transaction as more requests come in.

The TransactionWorker must keep track of the partitions that queries have run on for a given transaction in order to commit the transaction on these partitions. The state of the operation is also maintained as a sanity check. For a given transaction id, the client can only send one query at a time and must wait for results until sending the next query. Therefore, the state is checked to ensure that the client is behaving correctly. Throughout the implementation, there are many assertions that check that the components are behaving as expected.

TransactionState

TransactionState is an object created to maintain the state that a TransactionWorker must know in order to execute a query for a given transaction correctly. Since the TransactionWorker persists for the duration of one query in a transaction, a separate object is used to maintain the transaction state. The transaction state is maintained in a mapping of id to state for all running transactions. In this design, the thread pool used to run TransactionWorkers can be smaller than the number of open transactions. The TransactionWorker grabs the correct TransactionState object for a given transaction id from the central mapping, which is synchronized on the mapping to ensure no race conditions.

RouterLoader

The RouterLoader loads the router from the database. The user must define which table-column pairs to build a look-up table over. The RouterLoader connects to each back-end and runs queries over the desired column for each table to populate the look-up table with the correct information. The reason for implementing the loading of look-up tables in this way is to ensure that the look-up table is consistent with the back-end data at start-up. In addition other configurations such as hash and range partitioning can be supported and specified.

LookupTable

LookupTable is an interface which defines the abstract idea of a look-up table and how to interact with a look-up table. The key function of this interface is getPartitionMap which accepts a parsed statement and returns a mapping from partition id to the query to be run on that partition. Two classes inherit the LookupTable interface:

- IntLookupTable - A look-up table over integers which uses the Colt int-int hash map [5] as the storage engine for the look-up table.
- BasicLookupTable - A look-up table using the Java native hash map as the storage engine for the look-up table. This look-up table supports integers and strings as the column type that the look-up table is built on.

An alternative look-up table implementation that is not included in the current version of the router is an array look-up table. In an array look-up table each index of the array holds the partition information for a tuple with that id. Therefore, arrays would work well for dense data. In Chapter 7, I compare the performance of different look-up table implementations.

ConnectionWorker

The ConnectionWorker connects to a single MySQL back-end. It receives work on a blocking queue as input and adds results to a blocking queue as output. The TransactionWorker sets the output queue that the ConnectionWorker places results on so that the TransactionWorker can receive the results. The TransactionWorker starts processing results as they are received.

5.3 Thread Model

Figure 4-1 shows how the data is passed between components. The multiple layers of threads (TransactionWorkers, ConnectionWorkers) is necessary to avoid bottleneck and allows for queries from multiple clients to be run at the same time. Figure 5-1 shows how data is passed over time between the components. TW1 and TW2 are two different TransactionWorkers. CW1, ..., CW5 are the different ConnectionWorkers in use. This diagram illustrates two

different queries processed and answered simultaneously. The red, downward arrows denote work being done. CW1 and CW2 are doing work for TW1. CW3, CW4, and CW5 are doing work for TW2. The diagonal arrows denote the passing of work between the components.

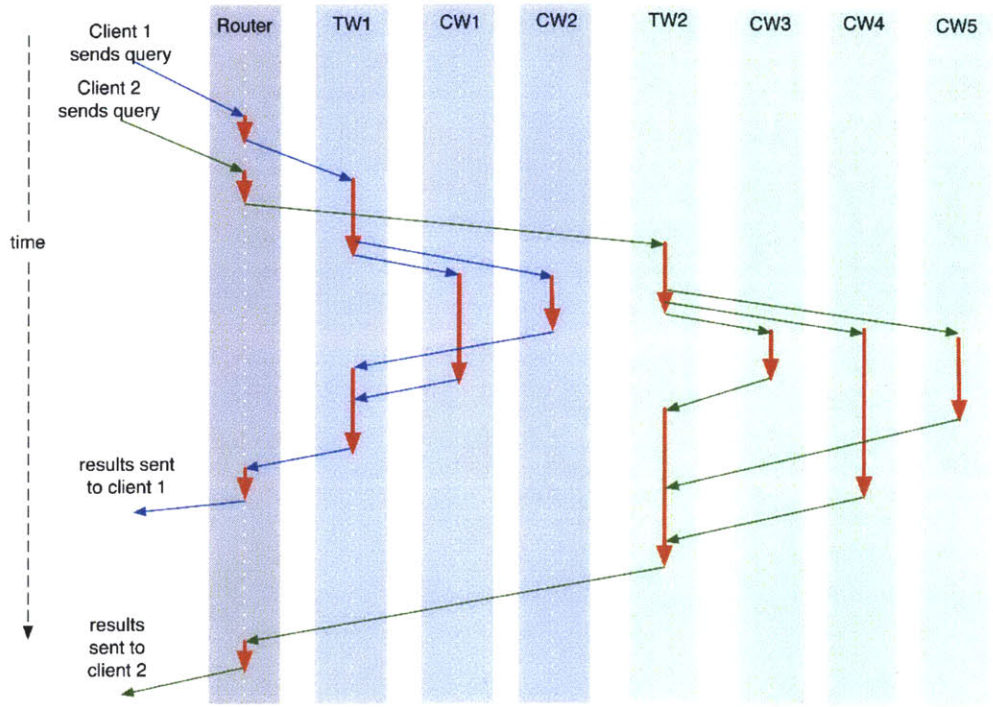

Figure 5-1: Thread diagram of asynchronous distributed queries being executed across different threads. The Router receives requests from clients and passes work to the TransactionWorkers (TWs). The ConnectionWorkers (CWs) send the queries to the back-ends.

Chapter 6

Experimental Evaluation

In this chapter, I outline the experiments run using this distributed execution engine and router. I start by introducing the framework for testing. Then I show scale-out with synthetic data. Finally, I show a performance win on Wikipedia data.

6.1 Benchmarking Tools Implementation

I use throughput as a measure of system performance. The reason for this is that as databases scale with more data and users, the number of requests increases. For this reason, the database system needs to scale to handle a large rate of incoming queries. Specifically, I compare the maximum throughput different schemes can achieve.

To measure throughput, I add load on the router by adding clients until the throughput stops increasing. In order to get a full load on the back-end database, up to 500 clients are used to drive the router. The clients run on the same machine as the router in these tests.

6.2 System Configuration

In the following experiments, the back-end machines consists of a cluster of machines with the configuration outlined in Table 6.1. The software configuration is outlined in Table 6.2. The front end machines run the router and the client driving the router with a workload. In many scenarios, one front-end machine is sufficient to drive the back-end machines. In such a scenario front-end 1 is used. In the cases where 1 front-end is not sufficient, then front-end 2 is used in conjunction with front-end 1.

Machine	# CPUs	# Cores per CPU	CPU Model	clock speed (GHz)	RAM (GB)
back-end node	2	1	Intel Xeon	3.20	2
front-end 1	2	4	Intel Xeon E5530	2.40	23
front-end 2	2	4	Intel Xeon E5520	2.26	23

Table 6.1: Hardware statistics for experiments.

Machine	Operating System
back-end node	Linux 2.6.31-22-generic-pae
front-end 1	Linux 2.6.35-25-server
front-end 2	Linux 2.6.35-25-server

Table 6.2: Software statistics for experiments.

Each front-end machine is connected to the back-end machines via a single gigabit Ethernet connection and a switch to the individual back-end machines.

6.3 Modeling the Benefit of Fine Grained Partitioning

Because the main benefit of fine grained partitioning is to reduce the number of distributed operations, I begin with an experiment to measure overall system throughput. The experiment uses synthetic data and varies the percent of distributed queries in a workload between 0% and 100%. This test shows that distributed queries are more costly than non-distributed queries for OLTP-style workloads.

6.3.1 Effect of Adding More Back-ends

Experiment Setup

In this experiment, the database consists of 20,000 tuples with unique primary keys ranging from 0 to 19,999. The construct for each table is the following:

```
CREATE TABLE "info" (
  "id" int(11) NOT NULL,
  "data" varchar(255) DEFAULT NULL,
  PRIMARY KEY ("id")
) ;
```

This experiment tests the effect of adding more back-ends to support the same data set. Each back-end is one of the back-end machines as shown in Tables 6.1 and 6.2. Each back-end is limited to one CPU in order to reduce the work needed to be done by the router to fully load the back-end.

In the four back-end test, each back-end is loaded with one quarter of the data (5,000 tuples). In the eight back-end test, each back-end is loaded with one eighth of the data (2,500 tuples). Each tuple's data field is set to a 200 character string in order to mimic the size of a tuple in a real-world database, such as Twitter.

This test varies the fraction of distributed queries across the nodes. An example query for this test is something along the lines of:

```
SELECT COUNT(*) FROM info WHERE id IN (5, 555, 2424, ...);
```

The reason for selecting the `COUNT` is to reduce the network load. Selecting the payload in the current configuration causes a bottleneck on the network side rather than on the back-end's CPU capabilities. Fortunately, in a true distributed database, ten gigabit Ethernet connections could be used and each back-end could have a dedicated Ethernet connection to the router. The `COUNT` query does approximately the same work as a `SELECT *` query since it has to verify the ids exist and then sum up the number of results, but it reduces the payload to the router dramatically.

For this experiment, 80 different ids are in the `IN` clause. This is analogous to requesting information for a user's friends such as their names or statuses. 80 ids is comparative to the number of friends a user is likely to have in a real world web application.

Each client in the test repeatedly sends auto-commit queries of the above form to the distributed database. Once a query returns, the client immediately issues another query. When a client generates a distributed query, it is sent to each of the back-ends. For the four back-end case, 20 ids will be found on each of the four back-ends. In the eight back-end case, 10 ids will be found on each of the eight back-ends. In the case of a non-distributed query, all ids will be found on one back-end. The target back-end is varied so that each back-end sees approximately the same load.

The percent of distributed queries sent to the nodes varies in this experiment. For example, if the percent of distributed queries is 50%, then half of the queries are sent distributed and half of the queries will be non-distributed. In the four back-end case, one

forth of the non-distributed queries will be found exclusively on back-end 1, one forth will be found on back-end 2, and so on. Figure 6-1 shows the example breakdown for the scenario that the percent distributed queries is 50%.

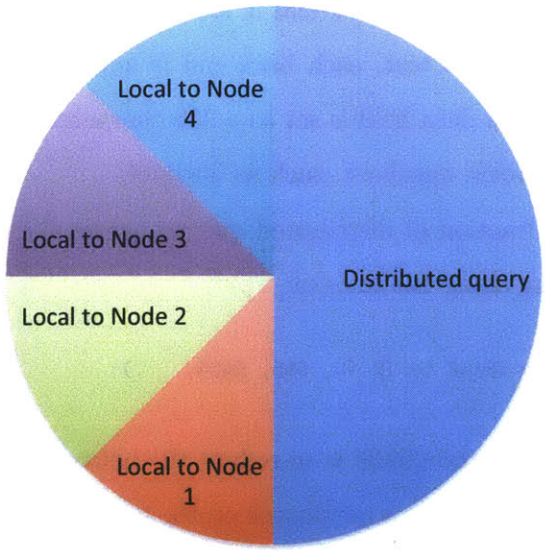

Figure 6-1: Example query breakdown for percent distributed = 50% on four back-ends.

Results

As the number of back-end nodes responsible for data increases, the throughput increases. Figure 6-2 shows the percent of distributed queries on the x-axis and the throughput in queries per second on the y-axis. The target max for eight partitions is computed by driving all eight partitions at 100% load directly and not through the router. We see that the eight partition performance for 0% distributed transactions performs close to the ideal case. The eight partition performance is 94% of the ideal performance expected. The reason for this will be elaborated on in the Wikipedia experiment section, but it is likely attributed to the difficulty of keeping 100% load on all back-ends.

This experiment shows that by partitioning the data and increasing the number of partitions we can achieve ideal speedup as we add back-ends. The eight partition case performs a factor of two times faster than the four partition case at 100% distributed queries. When all of the queries are distributed, the eight partition scheme performs 2.1

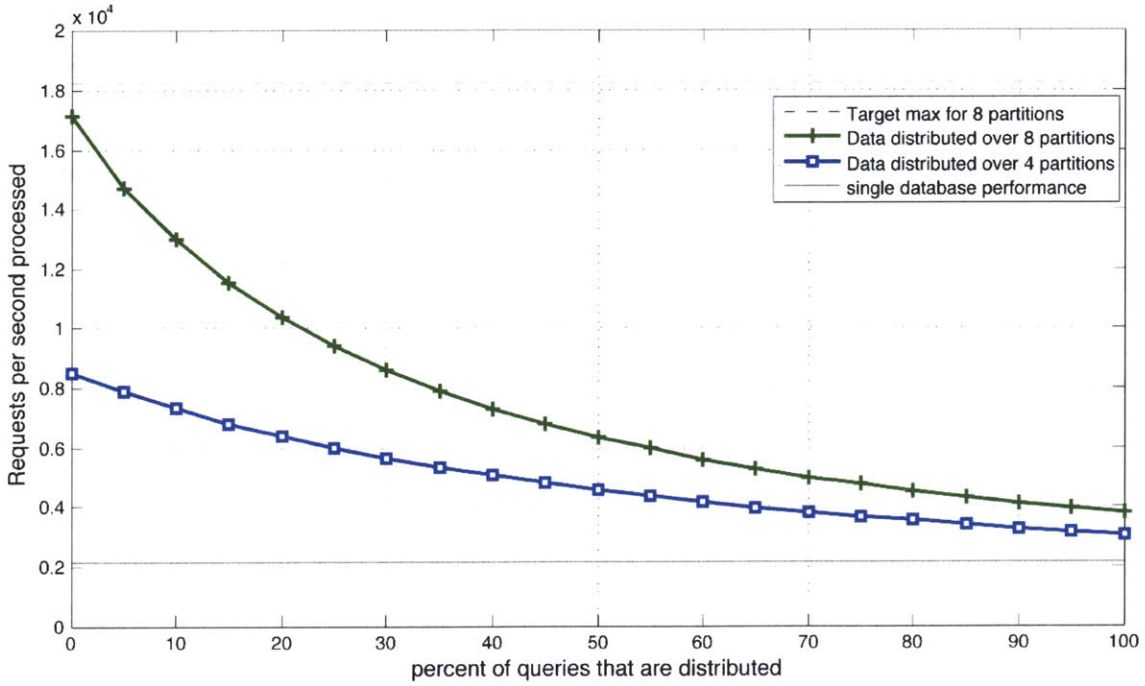

Figure 6-2: Comparing eight and four nodes for a distributed database's back-ends: Request rate (queries per second) vs percent of queries that are distributed.

times faster than a single database and the four partition scheme performs 1.7 times faster than a single database. In the eight partition case, the database with 0% distributed queries performs a factor of 4.5 times faster than the 100% distributed case. In the four partition case, the database with 0% distributed queries performs a factor of 2.8 times faster than the 100% distributed case. This supports that distributed queries have a high cost.

In summary, these results show that adding more back-ends scales out linearly when there are 0% distributed transactions. This supports that, if an ideal partitioning can be created, look-up tables can provide linear scale-out.

6.3.2 Effect of Fan-out of Distributed Transactions

Experiment Setup

This experiment uses the same database setup as Section 6.3.1, although the test is only over four partitions. This again has the same `IN` query selecting 80 tuples. The percent of distributed queries varies between 0% and 100%. Again, for a non-distributed query, the query requests ids from one back-end and hence goes to one back-end. For a distributed query, the query request ids from varying numbers of back-ends depending on the test.

The distributed query either requests ids that will be found on four, three, or two of the back-end nodes. A query that requests ids over all four back-end nodes requests 80 ids, 20 of which are located at each back-end. For a distributed query that goes to three back-ends, 27 ids are located on two of the nodes and 26 on another. Zero ids are found on a fourth node. For a distributed query sent to two nodes, 40 ids are found on one node and 40 on another node.

Results

Figure 6-3 shows the results of changing the measure of how distributed a distributed query is on throughput. From these results we can see that queries that touch fewer nodes run at a higher rate as compared to queries that touch all of the back-ends, even for the same percent of distributed queries. It is also important to note that there is still a significant cost to distributed queries even if they only touch two back-ends.

In the case of 100% distributed queries, the distributed query test accessing two back-ends per request only performs a factor 1.4 times fast than the distributed query accessing all of the back-ends. In the case of distributed queries accessing two back-ends the 100% distributed case performs approximately half as fast as the 0% distributed case.

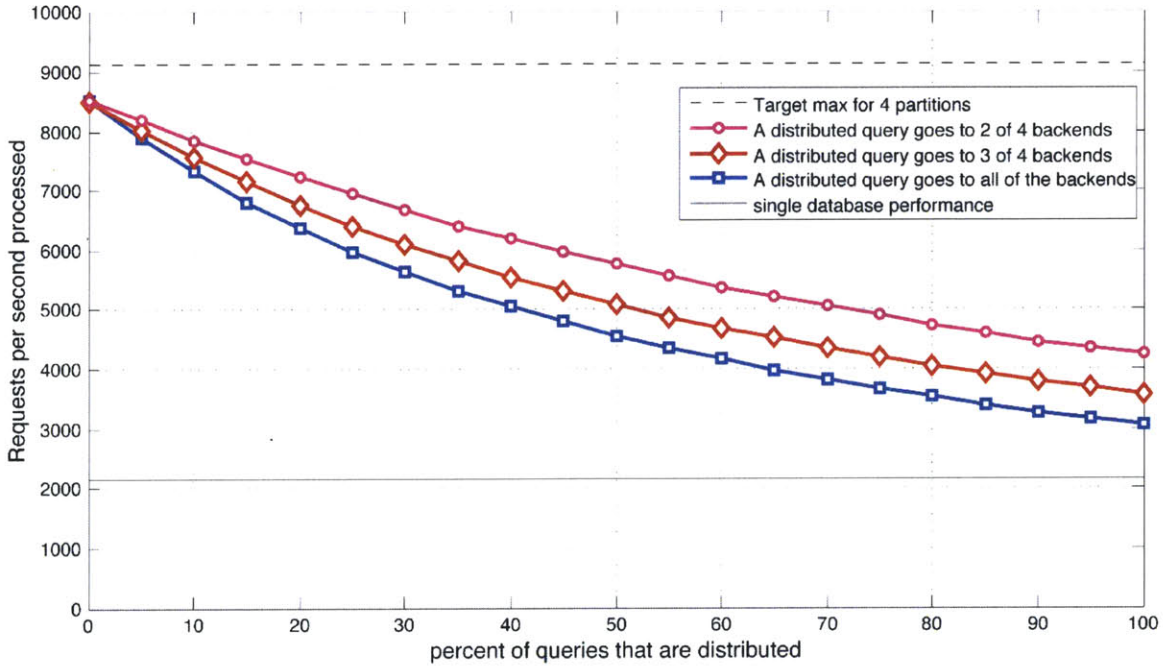

Figure 6-3: Effect of changing how distributed a transaction is: Request rate (queries per second) vs percent of queries that are distributed.

6.4 Wikipedia Example

In this section, a subset of Wikipedia obtained from MediaWiki is used. The goal of this experiment is to evaluate scaling on a real-world data set.

6.4.1 Wikipedia Data

Tables

The Wikipedia data set consists of three tables: `page`, `revision`, and `text`. The data is made publicly available my Mediawiki [29]. The following is the table structure that was used for the experiments. The important columns are listed while others are left out for brevity.

```
page : { page_id PK | page_namespace | page_title | page_latest REF revision.rev_id | ...}
revision : {rev_id PK | rev_page REFS page.page_id | rev_text_id REF text.id | ...}
text : {id PK | text | page REF page.page_id | ...}
```

Workload and Data

The data set has 100,000 entries in the page table, approximately 1.5 million entries in the revision table, and approximately 1.5 million entries in the text table and was extracted from Mediawiki, the software which powers Wikipedia. The database used totals about 36 GB of data.

An example transaction to look up the content of a page is as follows:

```
BEGIN;

SELECT page_id FROM page
   WHERE page_namespace = 1
   AND page_title = 'MIT_Electrical_Engineering_and_Computer_Science_Department'
   LIMIT 1;

SELECT * FROM page, revision WHERE (page_id=rev_page)
   AND rev_page = 12760945 AND page_id = 12760945
   AND (rev_id=page_latest) LIMIT 1;

SELECT length(text), flags FROM text WHERE id = 151111233 LIMIT 1;

COMMIT;
```

The reason for selecting the `page_id` and then doing a join with the `page_id` on `page` and `revision` has to do with the Mediawiki and how Wikipedia caches data among layers. I attempted to stay as close as possible to the Wikipedia workload for testing purposes. The final query would most likely select the text or some other meta-data from the `text` table. In order to reduce network bandwidth, I simply select the length of the text. The computation on the back-end nodes is similar to selecting the whole text since the MySQL back-end still needs to pull the text into memory to perform the computation. The difference is that a much smaller amount of data is sent over the network, which allows me to saturate the CPUs of the back-ends.

The first query extracts the `page_id` which the second query uses to find data about the page, including the id for the `text` table, which is used in the third query.

For the Wikipedia experiment, the above transaction is used as the workload to drive the database. An actual trace of what users selected in a Wikipedia log was used as the workload for selecting pages [26]. In this trace, more popular pages are selected more often.

6.4.2 Hardware Setup

For this test, I varied the number of back-end machines. Each back-end machine has the hardware setup described in Table 6.1 and the software configuration in Table 6.2. The back-end machines are limited to using one CPU. The front-end machines are described in the same table. Front-end 1 was used to drive the clients and the run the router.

6.4.3 Partitioning Schemes

Look-up Table Smart Partitioning

Since the workload performs look-ups by `page.page_id`, `page.page_title`, `text.id`, and `revision.rev_page`, the following scheme for partitioning the data was used:

- Range partition page on `page_title` such that accesses over title are approximately uniform across back-ends.
- Create a look-up table on `page.page_id` so that the router knows where a page is located based on `page_id`.
- Partition `revision` on `revision.rev_page` so that revisions where `revision.rev_page`

= `page.page_id` are co-located on the same partition to make the join in the second query local.

- Partition `text` on `text.id` so that `revision.rev_text_id = text.id` are co-located on the same machine.

This scheme ensures that the router knows where the data is on the back-ends and that all transactions will happen on one node. Based on the micro-benchmark in Section 6.3, we would expect to see this plan for partitioning and look-up tables out perform a hash-based partitioning of the Wikipedia data-set.

Hash Partitioning

The hash partitioning scheme was chosen to make joins for the second query run on one back-end. The scheme is as follows:

- Hash `revision` on `rev_page` and `page` on `page_id`, and co-locate those hashes such that `page.page_id` and `revision.rev_page` are located on the same back-end node.
- Hash `text` on `id`.

In the scenario of hash partitioning, the router no longer knows where to send query 1, so it must send it to every partition. Query 2 can then be sent to one partition. Query 3 can also be sent to one partition, which could be the same partition as query 2 but may not be.

Measuring Hash Partitioned Database Performance

Hashing is simulated by using look-up tables. Since latency was not a key measure of performance (as explained in Section 6.1), and the added latency of the look-up table is not significant compared to the network round trip time plus query answering time, this is a fair measure.

For the look-up table experiment, the look-up tables are generated from the data and loaded at start up time. Then multiple clients drive the router with the predetermined workload.

For the hash partitioned experiment, the hash partitioning was simulated using the router in order to keep the router implementation as similar as possible between tests. The

first query is sent to every location as opposed the location known to contain the page based on title, since the hash partitioning scheme would not have knowledge of partition based on title. The second query is sent to the correct back-end, which the router can determine via hash partitioning. Likewise, the third query is also sent the correct back-end, which the router can determine via hashing.

6.4.4 Growing the Database - Constant Partition Size

In this test, hash partitioning is compared to the smart partitioning with look-up table routing as the number of partitions grows to accommodate more data. This experiment tests the effect of throughput of transactions as the number of back-ends increases. In this particular experiment, the number of tuples in each back-end remains the same. Figure 6-4 shows the results of this test which compares the effect of adding partitions.

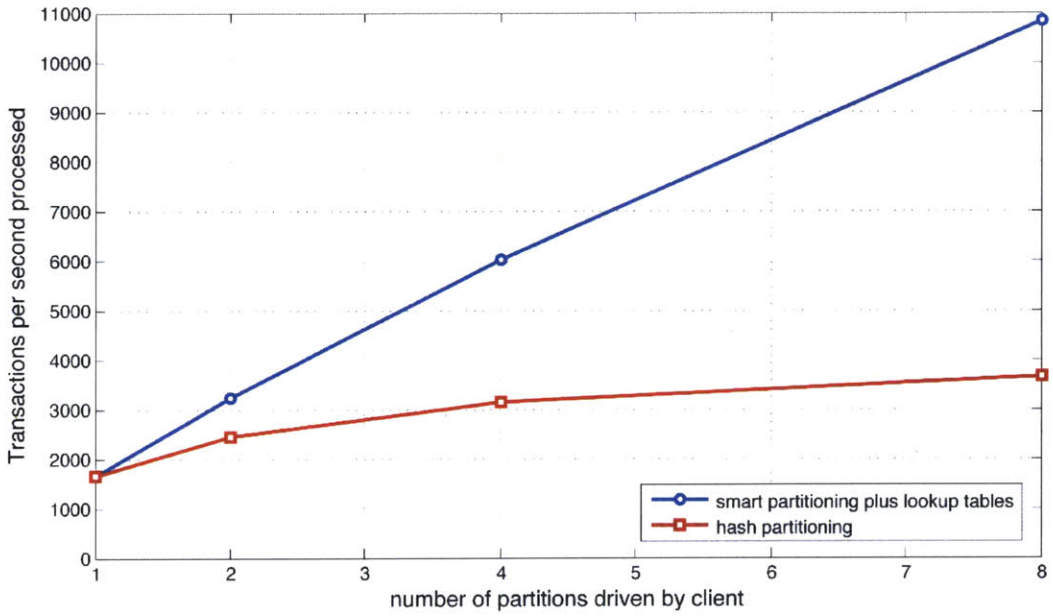

Figure 6-4: Effect of growing a database with more back-end nodes: Transactions per second vs Number of Back-ends. Each back-end has 1/8 of the total Wikipedia data.

Eight partitions with smart partitioning provides throughput that is 6.6 times faster than a single back-end. For hashing, eight partitions only yields 2.2 times faster performance verses a single partition. In the case of eight partitions, the look-up table implementation performs 3.0 times faster on throughput than hash partitioning.

It is also important to note in Figure 6-4 that the smart look-up table implementation

is continuing to scale at a near-linear rate, while the hash partitioning scheme has started to taper off. We would expect throughput to continue to grow as partitions increase for larger numbers of partitions and start to taper off much later.

6.4.5 Adding Back-ends - Constant Database Size

In this experiment, the total database size is kept constant while the number of partitions is varied. Figure 6-5 shows the performance of the system measured using throughput versus the number of back-ends used. The eight partition data point in Figure 6-5 is the same test as the eight partition data point in Figure 6-4. Eight back-ends' throughput is 5.9 times faster than a the single database performance for the smart partitioning. Performance for eight back-ends is only 2.0 times faster for the hash partitioning as compared to the single database. In the case of eight back-ends, the look-up table implementation performs 3.0 times faster on throughput as compared to the hash partitioning scheme.

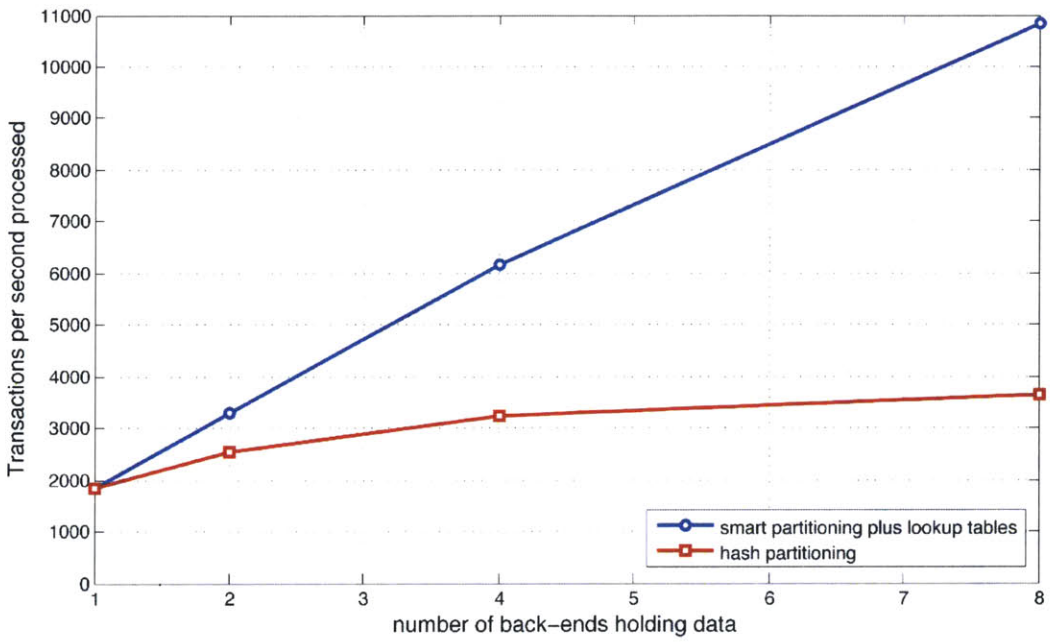

Figure 6-5: Effect of changing the number of back-ends storing the Wikipedia data. Transactions per second vs Number of Back-ends. Each configuration has the entire data set.

6.4.6 CPU usage for Wikipedia Example

In the previous section describing the results of 8 back-ends versus a single database, we would expect approximately an 8 times speed-up. Unfortunately there is only a 5.9 times

speedup. This is attributed to not completely utilizing the entire CPU of each back-end machine. The root cause to this is still unknown. One reason may be that the requests to the back-ends come in bursts. If there is a burst of requests that must be answered on partition 1, then new requests will not arrive at the other partitions until these are answered. If this is the case, it may be possible to perform live migration by moving data based on the current workload in order to keep all machines working at 100% CPU usage.

Another possibility for under utilization of some back-ends is that the Java implementation and network protocol may be batching requests in the underlying implementation. If this is the case, it may be possible to remove this problem by fine-tuning the system.

Figures 6-6 shows the CPU utilization of the eight CPUs from the eight back-ends for the smart partitioning. In this figure, some of the CPUs are underutilized. It may be possible to tune the system in order to max out the back-ends. In this scenario, the look-up table scheme could perform better than the results presented above. Figure 6-7 shows the same data for the hash partitioning scheme. This test takes three times longer to complete so the two graphs are on different x-axis scales. In the hash partitioning scheme, the CPUs are almost always running at 100%.

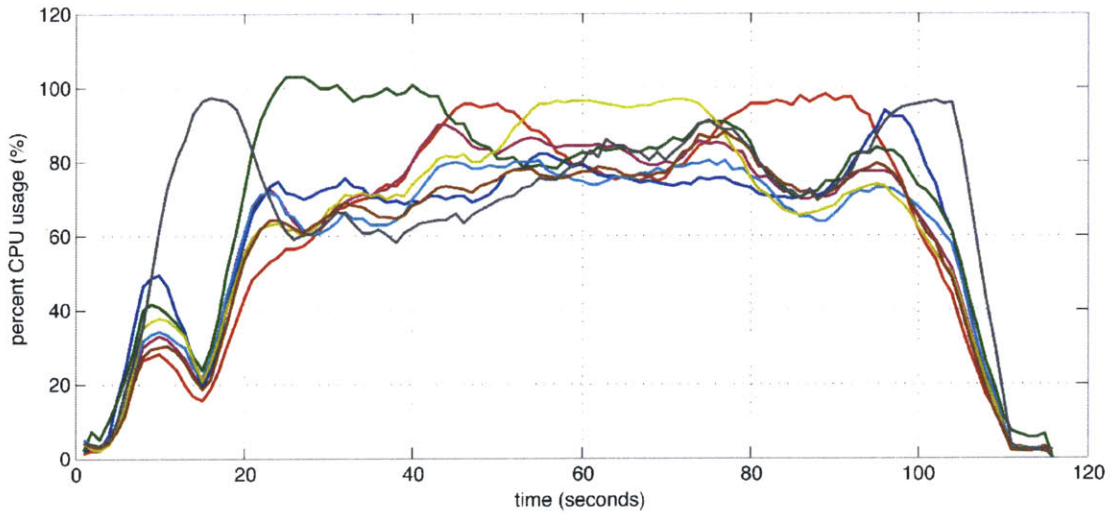

Figure 6-6: CPU usage for Wikipedia data set experiment using the look-up table scheme: Percent of CPU used vs time for the eight partition Wikipedia data set. Each line represents data from a different back-end CPU.

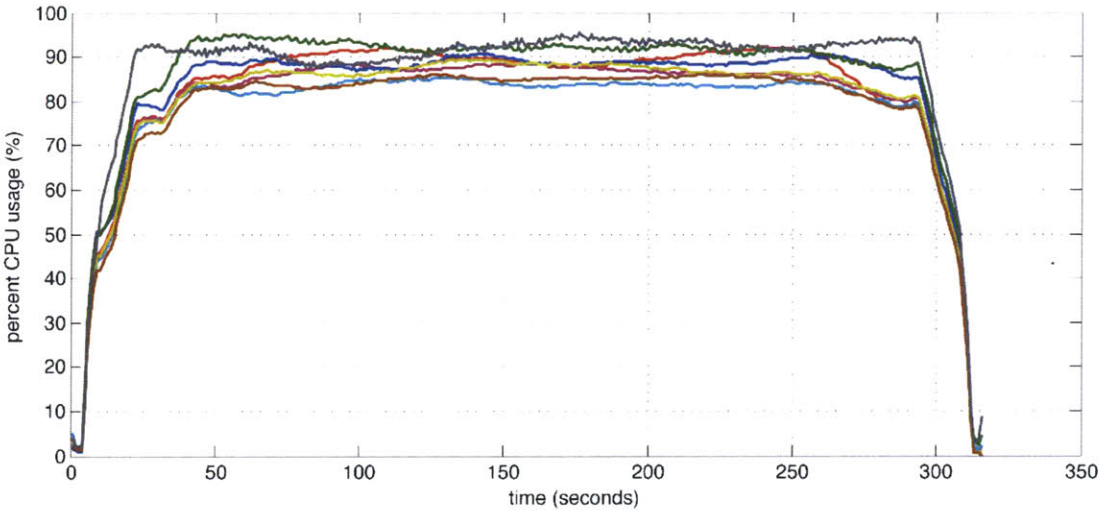

Figure 6-7: CPU usage for Wikipedia data set experiment using the hash-based scheme: Percent of CPU used vs time for the eight partition Wikipedia data set. Each line represents data from a different back-end CPU.

Chapter 7

Scaling Look-up Tables to Large Data Sets

One disadvantage of look-up tables as compared to hashing is that look-up tables must be stored somewhere in memory. This means that look-up tables often grow linearly with respect to memory as the database size grows. Another is that look-up tables must perform a memory look-up to determine the tuple partition as opposed to a simple hash computation.

In this section, I explore the memory usage and performance of look-up tables in Java. I also discuss a methods for scaling look-up tables by storing more important data.

7.1 Choosing the Correct Look-up Table Implementation

In choosing the correct look-up table implementation, it is important to minimize memory usage and maximize throughput of accesses. In the implementation described above, the Colt HashMap from the Colt Scientific library was used for integer look-ups and Java's native HashMap was used to String look-ups. Arrays were not used, but arrays are faster than map look-ups and can take up less memory if the keys are dense.

Figure 7-1 shows the effect of look-up table size on throughput of look-ups and memory utilization. As a reference, hashing a value needs zero memory and is also shown on the throughput graph to show how look-ups compare to hashing in speed. In this figure, the array is completely dense, which means that the length of the array is the same as the table size. In the scenario that the array is not completely dense, then the array size grows with inverse of density. Density can be quantified as the range of values divided by the number

of values, since the array needs to allocate space for a tuple value regardless of whether or not it is present in the database. For example an array that is 50% dense will require twice as much space to hold the sparse data. In real data sets, it is normally the case that the density of keys is close to 100%. This is because primary keys are normally auto-increment fields and data is not deleted often. Even if data is deleted, many database applications do an occasional id reshuffling. For example, in a Twitter data set obtained in September 2009, the user table was 84% dense supporting that tables over primary keys are often dense [6].

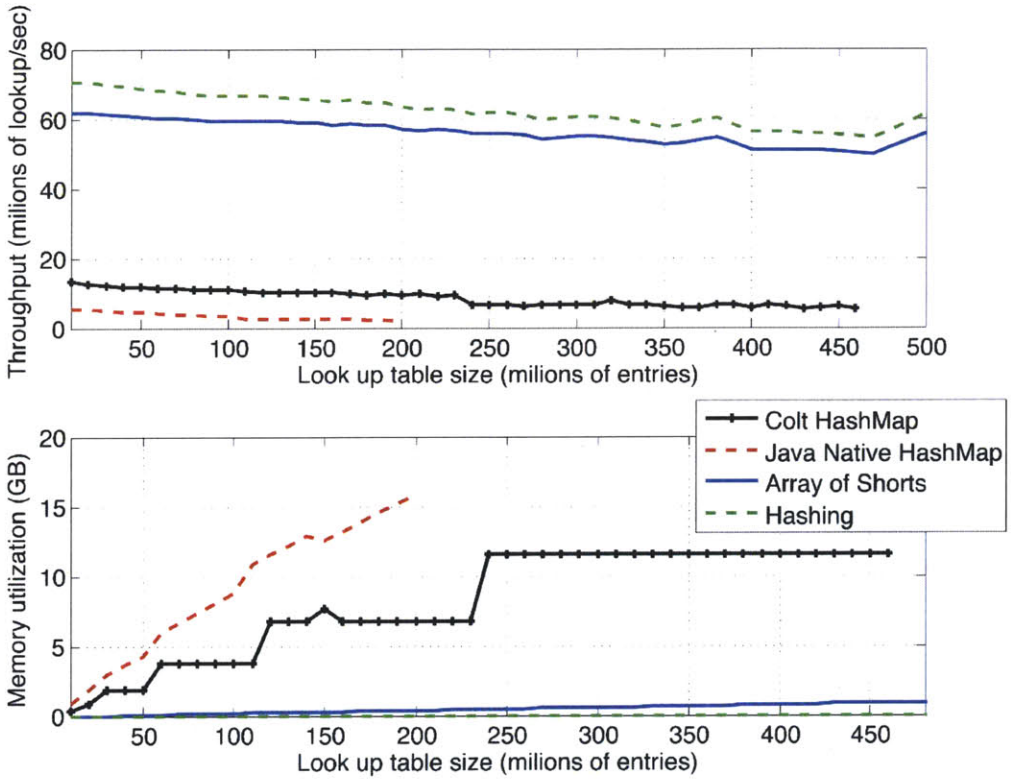

Figure 7-1: Look-up table performance: Throughput vs Look-up Table Size (top) and Memory usage vs Look-up Table Size (bottom)

From Figure 7-1, it is clear that arrays are better than both hash map implementations for dense keys from a memory point of view. We account only for the cost of selecting the correct partitioning, ignoring parsing and network costs (which are the same for every kind of partitioning). The top part of the figure shows throughput in millions of operations (look-ups) per second. Here, the array based implementation outperforms the two HashMap implementation by a significant amount (especially for large numbers of entries), and it is very close in performance achieved by a very simple modulo-based hashing. In either

case; however, millions of look-ups happen per second, suggesting that look-ups will not significantly impact latency.

7.2 Look-up Table as a Cache

The look-up table as a cache scheme allows restricting the size of the look-up table to be a fraction of the size required to hold a look-up value for every tuple. Specifically, I propose only storing the recently requested tuples in the look-up table. If the location of the tuple is not in the cache, then the query must be sent everywhere. After it is found, it can be added to the cache.

For the example of the Wikipedia data set used with an actual trace of requests show that 90% of the requests over the revision table and text table represented only 1.6% of the total data. The reason for this is that the revision and text tables hold every revision and the text for every revision for every page. Queries over revision and text normally request the newest version of the page. In addition, certain pages are much more popular than others. This type of skew in data accesses suggests that caching should perform well. This skew is present in many data sets and can be capitalized on in order to keep look-up table sizes down. For example, if we cache 1.6% of the data, 96GB of RAM could handle over one trillion tuples.

7.3 Hybrid Storage

Hybrid storage is similar to a cache, but instead of caching "hot tuples", the "hot" data is partitioned using fine-grained partitioning and the rest of the data is partitioned using hash partitioning. In this scenario, the look-up table can be much smaller than the data set and the router will always know the location of the tuples. The router first checks the look-up table for the tuple location. If it is found, then the location of the tuple is known and the query can be modified and sent. If the look-up table has a miss, then the value is hashed and the partition is still known and then the modified query is sent. The performance of this system is expected to perform close to the same as a pure look-up table scheme. The reason for this is that the location of a tuple is always known and the scheme can most likely be chosen to minimize the fraction of distributed transactions.

Chapter 8

Future Work

The look-up table implementation described in this thesis shows a proof of concept that look-up tables coupled with fine-grained partitioning can yield higher performance results as compared to hash-based partitioning. A few major components need to be built to make this prototype suitable for real world deployment. These are outlined below.

Distributed Query Planner

Missing in this research is a query planner that picks the best strategy for breaking a query into multiple queries for complex queries. In Section 4.1, I presented different joins plans that should be chosen based on the specific scenario. To make this choice automatically, information such as table cardinality and statistics on the database are needed. The planner would also have to have a cost model which would be more complicated than traditional planners for non-distributed databases. In addition it may be helpful for the planner to be able to plan both at the router level and the back-end nodes, specifying the plan all the way to the back-end nodes.

Two-Phase Commit

The implementation presented in this thesis supports transactions assuming the router and back-end nodes do not fail. To handle failures, a distributed commit protocol like two-phase commit is needed. Two-phase commit in the current version of MySQL is relatively slow. In order to implement two phase commit without a large performance overhead, modifications to the back-end node protocol are needed.

Chapter 9

Conclusion

Contributions

Scaling distributed transaction processing databases linearly is a significant real-world problem. I presented look-up tables as a novel way to support fine-grained partitioning of complex data that can yield linear speedups. My implementation and results show that fine-grained partitioning supported through the use of look-up tables is both feasible and one possible solution to the partitioning problem with growing database sizes. I used both real and synthetic data to show that look-up tables can address difficulties in scaling out a system when data relations are complex.

Results Summary

I show that a database can achieve near linear speed-up with scale out when ideal partitioning is achieved. I also show that fine-grained partitioning can be supported through look-up tables. Using a smart partitioning scheme on a Wikipedia data set, I show that my look-up table implementation performs three times faster than the best hash partitioning scheme using eight back-end machines. I also show promise that my scheme will continue to scale-out where hash partitioning hits a wall in performance.

Bibliography

[1] Amazon web services. `http://aws.amazon.com/`.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[3] Hari Balakrishnan. Cloud computing lecture 1, February 2011. `https://web.mit.edu/6.897/www/notes/L1.pdf`.

[4] Case studies. `http://aws.amazon.com/solutions/case-studies/`.

[5] cern.colt.map.abstractmap. `http://acs.lbl.gov/software/colt/api/cern/colt/map/class-use/AbstractMap.html`.

[6] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *In Proceedings of the 4th International AAAI Conference on Weblogs and Social Media (ICWSM).*

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *IN PROCEEDINGS OF THE 7TH CONFERENCE ON USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION - VOLUME 7*, pages 205–218, 2006.

[8] 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. Technical report, In Proc. 34th VLDB, 2008.

[9] Carlo Curino, Evan Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational Cloud: A Database Service for the Cloud. In *5th Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2011.

[10] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, pages 48–57, 2010.

[11] 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 twenty-first ACM SIGOPS symposium on Operating systems principles, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[12] Google docs. `docs.google.com`.

[13] Nick Kallen. Big data in real-time at twitter. `http://www.slideshare.net/nkallen/q-con-3770885`.

[14] Avinash Lakshman and Prashant Malik. Cassandra- a decentralized structured storage system.

[15] Bruce G. Lindsay. A retrospective of r*: A distributed database management system. *Proceedings of the IEEE*, 75(5):668–673, 1987.

[16] Memcachedb. `http://memcachedb.org/`.

[17] Mongodb. `http://www.mongodb.org/`.

[18] Protocol buffers - google's data interchange format. `http://code.google.com/p/protobuf/`.

[19] Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikolaos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine(s) that could: scaling online social networks. In *SIGCOMM*, pages 375–386, 2010.

[20] Josep M. Pujol, Geogos Siganos, Vijay Erramilli, and Pablo Rodriguez. Scaling Online Social Networks without Pains. NetDB 2009, 5th International Workshop on Networking Meets Databases, co-located with SOSP 2009, October 2009.

[21] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[22] Stephen Shankland. Amazon cloud outage derails reddit, quora, April 2011. `http://news.cnet.com/8301-30685_3-20056029-264.html`.

[23] Maggie Shiels. Twitter co-founder jack dorsey rejoins company, March 2011. `http://www.bbc.co.uk/news/business-12889048`.

[24] Sql azure. `http://msdn.microsoft.com/en-us/windowsazure/sqlazure/`.

[25] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[26] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. `http://www.globule.org/publi/WWADH_comnet2009.html`.

[27] Amin Vahdat. Presentation summary "high performance at massive scale: Lessons learned at facebook", November 2010. `http://idleprocess.wordpress.com/2009/11/24/presentation-summary-high-performance-at-massive-scale-lessons-learned-at-facebook/`.

[28] Where are my files stored?, May 2010. http://www.dropbox.com/help/7.

[29] Wikimedia downloads. http://dumps.wikimedia.org/.