ARIADNE: A NOVEL HIGH AVAILABILITY CLOUD DATA STORE WITH TRANSACTIONAL GUARANTEES

by

Alexander G. Connor

B.S. in Computer Science, University of Pittsburgh, 2008

Submitted to the Graduate Faculty of

the Department of Computer Science in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2012

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This thesis was presented

by

Alexander G. Connor

It was defended on

December 3, 2012

and approved by

Alexandros Labrinidis, Associate Professor

Panos K. Chrysanthis, Professor

Demetris Zeinalipour, Lecturer, University of Cyprus

Thesis Advisor: Alexandros Labrinidis, Associate Professor

Panos K. Chrysanthis, Professor

ARIADNE: A NOVEL HIGH AVAILABILITY CLOUD DATA STORE WITH TRANSACTIONAL GUARANTEES

Alexander G. Connor, M.S.

University of Pittsburgh, 2012

Modern cloud data storage services have powerful capabilities for data-sets that can be indexed by a single key – key-value stores – and for data-sets that are characterized by multiple attributes (such as Google's BigTable). These data stores have non-ideal overheads, however, when graph data needs to be maintained; overheads are incurred because related (by graph edges) keys are managed in physically different host machines. We propose a new distributed data-storage paradigm, the key-key-value store, which extends the key-value model and significantly reduces these overheads by storing related keys in the same place. We provide a high-level description of our proposed system for storing large-scale, highly interconnected graph data – such as social networks – as well as an analysis of our key-key-value system in relation to existing work. In this thesis, we show how our novel data organization paradigm will facilitate improved levels of QoS in large graph data stores.

Furthermore, we have built a system with our key-key-value system design – Ariadne – that is de-centralized, scalable, lightweight, relational and transactional. Such a system is unique among current systems in that it provides all qualities at once. This system was put to the test in the cloud using a strenuous concurrent workload and compared against the state of the art MySQL Cluster database system. Results show great promise for scalability and more consistent performance across workload types in Ariadne than in MySQL.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

LIST OF ALGORITHMS

PREFACE

For Jenn. For Trevor. For my parents. For Alex and Panos.

1.0 INTRODUCTION

The phenomenon of "SlashDotting" [1] occurs when a link to a website is posted on the eponymous news site, and subsequently visited by hundreds of thousands or even millions of users in a short time span. The result is either that the linked website crashes due to load that it has never before experienced, or it gracefully handles the increase, potentially attaining new levels of fame and success overnight.

Furthermore, the hypothetical website that has been SlashDotted is likely not merely a website, but a web application. This application likely requires user input, thus creating a potentially write-heavy workload for the data management back-end. This back-end must therefore not only be able to scale, it must be available for writes as well as reads under heavy load. As such, scalability – including write scalability – has become an area of great interest in the information technology community, especially in support of scalable data stores.

This is the case because, unlike a web server that can be duplicated ad infinitum without impacting the quality of data served, a database management system typically cannot be duplicated in the same way. The reason for this has been primarily that data has to be kept strongly consistent between all replicas, and the work of doing so makes the service too slow to be useful. In recent years, however, a movement has begun towards data management systems that provide a simpler feature set than that of relational database systems, but can indeed be scaled out to an arbitrarily large number of machines. This is achieved by trading strong consistency for special form of weak consistency, which we will describe below.

The earliest form of such a system was the key-value store, exemplified by Amazon's Dynamo [11]. Key-value systems provide a basic API that allows an application to put and get key-value pairs. Later systems have added features such as structured document storage,

[1] http://www.slashdot.com/

MapReduce support and application-specific query languages. These systems are yet non-relational and non-transactional, hence they bear the colloquial classification of "NoSQL Database".

Availability and raw performance are not enough for a dynamic web application – even if the web application is available and performs reasonably quickly under high load, users will expect its responses to be consistent. "NoSQL" systems to date do not provide strong consistency guarantess – they at best provide *eventual consistency* – the guarantee that after some finite period of time the data on any two replicas will be consistent. The trade-off of strong consistency for performance is typically motivated by the so-called "CAP Theorem", a result based off of Brewer's Conjecture. In this work, we show that this trade-off is erroneously made in many cases. Often, the CAP Theorem or Brewer's Conjecture are cited as reasons for compromising functionality and consistency in order to gain higher scalability. In this work we shall show that for the space of large (but not huge) installations, this compromise does not need to be made.

Another look at CAP: Suppose there is a distributed system that serves some data item X, replicated on hosts A and B. The Consistency-Availability-Partitioning Theorem (CAP for short) states simply that given a network partition between A and B, (the P in CAP), the system can either (1) continue to serve requests and present inconsistent data (thus being available – the A) or (2) stop serving requests until it can guarantee consistency again (the C). Thus, a system can only uphold two of the three qualities – guaranteed availability, guaranteed consistency, guaranteed funcionality in the presence of network partitions – at any time.

Note that the availability-consistency trade-off must occur *only* when there is a network partition. This is a special sort of failure, as it specifically means that some group of machines is unable to communicate with another group of machines, and both are able to serve requests. This happens most often in geographically distributed installations wherein the system consists of two or more separate data centers. These data centers could potentially be on different continents. In most medium-sized installations, the machines are housed in one or more data centers in the same locale. In these installations there are more than

a few machines, though – a number beyond the scope that RDBMS (Relational Database Management System) replication can handle.

In this kind of installation, a system needs to be fault-tolerant and partition-tolerant, but can afford to provide stronger consistency guarantees in the normal state. That is, when there are no network partitions. As some of these "NoSQL" systems have become publicly available, including MongoDB [26] and CouchDB [4], developers of web applications and services are beginning to choose these tools over traditional relational database systems. Oftentimes, though, these applications do not utilize multiple data centers, or massive installations. Although, in theory, the trade-offs described by the CAP theorem could exist for these services, they often do not exist in reality.

The purpose of this work is to provide a system that sits in this middle ground – it is intended for those services that need a larger data store than a single RDBMS server but are not so huge that data store hosts are geographically separated. Furthermore, when it can provide a consistency guarantee (this should be most of the time), it provides richer features while staying light-weight. These include a relational model and transactions – and the system is truly designed for the cloud, as it can operate on virtual machines with minimal resources.

Key-Key-Value Model: Emerging cloud services often work with requests for data identified by two keys – examples include Facebook (facebook.com) and Twitter (twitter.com). Consider Twitter, a social networking service where users send status updates and read the status updates of their friends, or people they *follow*. Twitter thus needs to be able to specify this user to user relation: `(user1, user2, relationship)`. For our example, suppose that *relationship* is `follows` and `alice` is the user who follows `bob`. The relationship could also be something else, such as `blocked`, where the first user is not allowed to see the second user's updates.

Users need to be able to ask queries such as *who follows* `bob`*?* and *who does* `alice` *follow?* To implement these relations – the *follows-list* and the *followers-list* – in a key-value store, both lists need to be maintained for both `alice` and for `bob`. If only the follows-list was maintained, the follows-list for every user in the graph would need to be examined when determining the followers-list for `bob` – this means a high-cost join. One could argue that edges of the graph could be indexed in an intelligent fashion; even with the best indexing the lists for `alice` and `bob` could still be stored on separate machines. This creates an overhead in performance or a potential inconsistency in the data [14].

We propose to unify these four lists by storing data as a `(source-key, target-key, value)` relation; now, if both `alice` and `bob`'s data are kept in the same place, then both queries can be answered from *one* site. Our proposed approach reduces the overhead of storage by eliminating duplicate records and reduces performance overhead and improves consistency by keeping related records in one site [7].

Working Example: Throughout this thesis we will discuss these concepts and our work within the context of a social networking web application (such as Facebook or Twitter, mentioned earlier). Our example social network supports interrelated data primitives of users, friendships, status updates, and photos. This is a fitting example as

1. Social networks have write-heavy workloads, due to users continuously posting status updates and photos
2. A social network on the rise will have a rapidly growing user base, and thus need to scale quickly and seamlessly
3. Interactions with the data store for social network pages often require joins – e.g. show photos from some user, show posts shared from one user to another, or show photos from one user's friends.

We will describe a minimal schema, user interface and queries to run against Ariadne for the social network application. This synthetic workload is designed to emulate that of a real social network.

Contributions: In this work, we introduce the Ariadne Distributed Data Store, a novel scalable data management system that contributes the following:

1. A simple SQL-like interface that supports basic joins
2. Transactional guarantees similar to snapshot isolation
3. A near relational model
4. A novel model for scalable data stores that extends the key-value model, the *key-key-value* store
5. A high-level design for the *key-key-value* system
6. An implementation of the Key-Key-Value store design

Roadmap: In the next chapter we will review related work from the literature. In Chapter 3 we will discuss the system design of Ariadne. In Chapter 4 we will discuss the implementation details. In Chapter 5 we will evaluate our system empirically. Finally, we conclude with Chapter 6

2.0 RELATED WORK

As we stated above in Chapter 1, our system provides a unique combination of scalability, transactional guarantees, low resource requirements, and a relational model. We shall examine the state of the art in research and production systems, and demonstrate this uniqueness. These systems fall into the following categories: *Cloud Data Stores*, *Distributed RDBMSs* and *Hybrid Systems*. Figure 2.0.1 shows a visual breakdown of the works mentioned here.

2.1 CLOUD DATA STORES

Some of the more well-known internal cloud systems include Dynamo of Amazon.com [11], PNUTS of Yahoo! [8] and BigTable from Google [5], which use key-value stores. Key-value stores manage data structures wherein there is a primary key and an object; the object can only be referred to by the key.

The creators of Dynamo and PNUTS were motivated by the observation that most of the data that applications work with are indexed only by a primary key and most often are accessed or modified one item at a time [11, 8]. This observation allowed them to design massively scalable systems – as a value is accessed independently of other values, sharding of data is trivially simple using a consistent hashing method. With the data sharded, replicas can be arbitrarily small, allowing the system to run on an unlimited number of fixed or similar-sized machines.

The reality is often, though, that data items are interrelated – the key-value model forces the application to manage these connections. In a traditional relational system an application could pose declarative join queries and expect a correct result within a predictable response

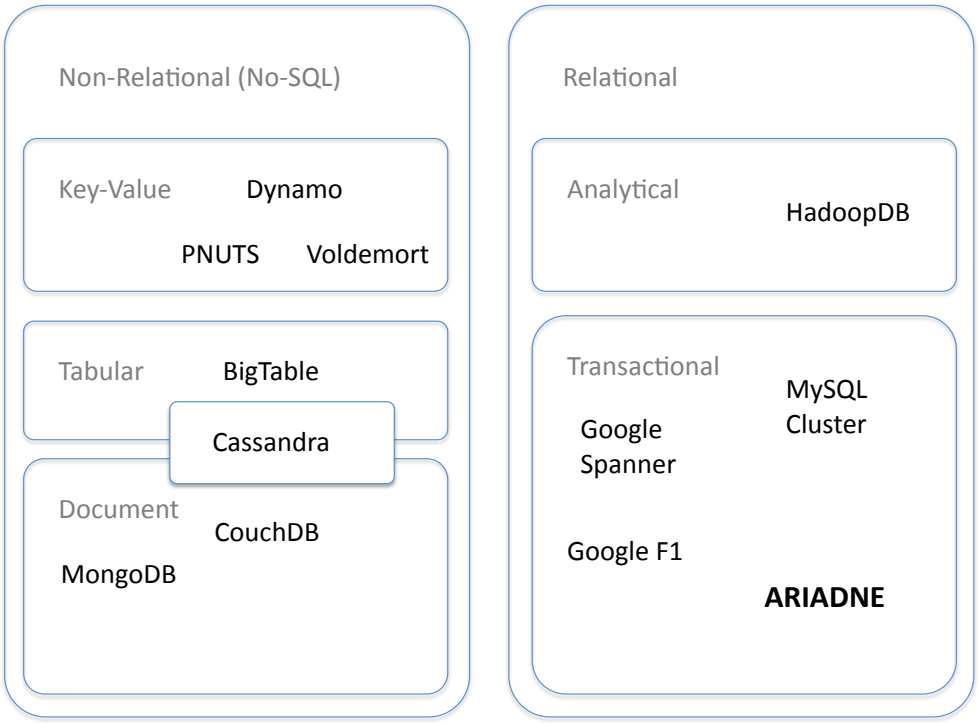


Figure 2.0.1: Production systems related to Ariadne

time. In a key-value store, however, the application must design the query plan itself, and implement the necessary join algorithms. Alternatively, the application could store all related data in a value – this would be, in effect, a denormalized schema – the pitfalls of which include the cost of maintaining all copies of a common data item.

Our work sits in between the key-value model and the relational model by providing the following: a model for associating keys, the key-key-value model [7]; join queries and query plans that leverage key-key-values; and transactional guarantees. It does not provide a complete relational model, full SQL, or true serialization – by making this compromise it is scalable in the same way as key-value stores.

The key-value model serves as the basis for several other current systems and designs. BigTable manages more complex data structures by allowing values to have many attributes – these are still identified by only one key. Furthermore, BigTable is not truly decentralized, whereas our system is. Also, BigTable provides no transactional guarantees, no join queries and no foreign key support – all of which Ariade does provide.

G-Store [10] and ecStore [31] are University prototypes of transactional key-value stores that support multiple key manipulations with strict consistency. Although these support operations on multiple keys, they still store data as key-values – our system, although it does not provide strict consistency, operates on key-key-value data as well as key-value data.

Examples of cloud data stores currently available to the public include Cassandra [1, 17], MongoDB [26], CouchDB [4] and Project Voldemort [2]. These systems provide highly available, high-performance data access. However, they are agnostic to the relationships between data and keys, which are assigned to hosts by a hashing function in the distributed setting. Our proposed system leverages the relationships between keys to create locality by storing related keys on the same host.

Cassandra is a distributed document store that leverages the key-value store technology of Dynamo and the simple data structure patterns of BigTable. It is at the time of this writing used in production systems by several prominent information technology firms. Cassandra scales out to large numbers of machines, which can be geographically distributed as well. It is designed to provide reliable access to data and fast response times, without any centralization or single point of failure [17]. Cassandra does not, however, provide transactional guarantees,

foreign keys or joins, as Ariadne does.

Similar in design to Cassandra is MongoDB [26] – also a highly scalable document store. Unlike Cassandra, though, MongoDB's only supports master-slave replication schemes – to handle failures, it uses a distributed election algorithm to create new masters. Partitioning of data across machines is also supported using a scheme similar to Cassandra, but requires administrative configuration. Our system and Cassandra provide this functionality transparently. Our system also supports joins and provides transactional guarantees, which MongoDB does not provide.

Another publicly available document store is CouchDB [4]. CouchDB supports true master-master replication, unlike MongoDB. It does not, however, directly support sharding – as such, sharding and dynamic scaling need to be managed by an outside tool. Ariadne is designed to support these features within itself.

2.2 DISTRIBUTED RDBMS

Approaching the other end of the spectrum, RDBMSs have begun providing facilities for high availability through replication and sharding. Notable commercial products that provide these features include Oracle [21] and MySQL Cluster [28]. Although we will examine several systems proposed in the literature, we will focus on MySQL Cluster as this has been used in our experimental evaluation.

MySQL Cluster utilizes three types of hosts:

1. Data nodes, which are only responsible for storing data
2. API nodes, which provide the interface with the end user – these are the points at which MySQL clients connect
3. A single management node that is responsible for disseminating configuration information to the other nodes

The structure of the installation needs to be decided in advance by an administrator and the management node is solely responsible for starting the cluster and managing changes to

the cluster's structure. The data nodes, once they have started, communicate directly with one another to achieve synchronous replication. Transactions are achieved using two-phase commit [27, 28].

This enviroment differs from Ariadne in three ways. Firstly, data in Ariadne is replicated asynchronously. Secondly, transactions use a novel form of optimistic two-phase commit (see Chapter 4 below for details). Thirdly, Ariadne does not distinguish between different node types; each node shares the responsibility of providing API, configuration and data storage. The loosened transactional guarantees and asynchronous nature allow Ariadne to provide a higher level of availability than MySQL.

Lau and Madden proposed a system in 2006 for a high-availability distributed data warehouse [20]. Our work uses a similar approach of timestamping data to maintain consistency. This system, however, is intended to be used as a data warehouse and as such does not support a write-heavy workload. Ariadne differs in that it is intended to be a highly dynamic data store, supporting frequent reads and writes. It also differs in the timestamping method – Lau and Madden require that the timestamp be relative to physical time. The commit protocol they use also differs from the one we use; ours is a less strict version of the classic two-phase commit than theirs.

Fischer and Michael, in 1982, proposed an algorithm for high availability distributed dictionary, which provides a best-effort level of data consistency [13]. This algorithm differs in its approach from our own, but makes the same fundamental trade-offs as our system and other current cloud data stores.

2.3 HYBRID SYSTEMS

In the analytical space, HadoopDB is a massive scale analytical DBMS [3]. HadoopDB translates SQL queries into MapReduce jobs and then executes those jobs to answer queries. It is, however, focused more on analytical (read-heavy) workloads, whereas Ariadne is focused on more mixed and write-heavy workloads.

Google uses internally a system called Spanner [9]. Spanner, one of the early hybrid "NewSQL" systems, is closely similar to Ariadne. It uses a near-relational model similar to Ariadne's, provides transactions, provides an SQL interface, and is designed for scale. There are several key differences, however. Most notably, Spanner is designed for a global, multi-datacenter installation. As such, it provides consistency by relying heavily on physical mechanisms (such as the atomic clock and GPS satellites). Ariadne has no dependency on physical systems, as it uses theorically strong methods for synchronizing events. Spanner transactions use two-phase commit, requiring a stonger level of consistency than Ariadne. Furthermore, Spanner uses hierarchies of servers and management servers, whereas Ariadne is truly decentralized.

Spanner is the foundation for Google's F1 distributed RDBMS [29]. F1 is more like a traditional RDBMS, as it enforces strict schemas and provides secondary indexes. Ariadne does not require a strict schema – it only requires that foreign key relationships be established in advance.

3.0 SYSTEM MODEL

Ariadne was designed with the following core principles:

- Each node has exactly the same responsibilities as every other node
- The cluster can be configured in real time
- Concurrency is managed primarily using message-passing queues

Ariadne is composed of the following major components:

- API
- Query Planner
- Transaction Manager
- Address Table
- Data Store
- Grid Manager

The components are connected by queues, and each has one or more worker processes that perform tasks within the component. The system also leverages a few global, concurrent data structures, including a data cache and an address table.

3.1 API AND QUERY PLANNING

Ariadne offers a simple form of SQL for its application interface. The following statement types are supported:

- `SELECT` – Given a list of columns, tables, and conditions return data from the store. Only one level of join is supported – subqueries are a goal for future work. Only equality joins are supported. Where conditions can only be equality conditions. An equality condition on the primary key of at least one table referenced in the join must be given.
- `UPDATE` – Update columns on a single table. A where condition on the primary key of the table must be given
- `CREATE LINK` – Create a foreign key relationship between columns. The relationship must always be between a non-primary key column on one table to the primary key column of another. There can only be one primary key column per table, the `id` column. The foreign key relationship must be in place for joins to work correctly between the linked table
- `DELETE LINK` – Destroy a given foreign key relationship. A foreign key column and parent relation must be given.

3.1.1 Data Model

The Ariadne data model is a simple relational model. An instance can have several relations; each tuple is identified by a single primary key. Within a tuple, each column has a unique name and a value. Note that Ariadne uses a loose schema; the schema of a relation is defined as the set of column names used in all of its tuples. A tuple implicitly has `NULL` values in those columns that it has not explicitly given a value to.

A tuple can contain any number of foreign keys to other relations; these must be defined in advance by the `CREATE LINK` command. These columns also implicitly have a `NULL` value. When a foreign key column has a value, that value is considered to be a primary key value for the foreign relation. If the value does not exist as a primary key in the foreign relation, then it implicitly points to a tuple in the foreign relation where each value is `NULL`.

3.1.2 Query Planning

The query planner takes a given query from a session, and does the following:

1. Initiate a transaction, if one does not exist yet.

2. Generate a physical plan from the SQL.
3. In the case of a `SELECT`, optimize the plan
4. Execute the query plan and return results
5. In the case of single statement, commit the transaction

3.2 TRANSACTION MANAGER

All activities run within the context of a transaction, including individual statements. This is because even a single select or update will touch several columns; in the case of a join, several rows from several tables could be drawn. The transaction manager on a global scope tracks active transactions that the node is aware of. Also, for each transaction, it provides an in-memory buffer for data used by the transaction. This buffer improves the performance of the query as well as maintains a snapshot of the data that the transaction can see.

The data model, within a transaction, is the following hierarchy:

$$\text{Transaction ID} \rightarrow \text{Schema} \rightarrow \text{Table} \rightarrow \text{Row ID} \rightarrow \text{Column} \rightarrow \text{Data} \tag{3.2.1}$$

Each transaction cache supports the following operations:

- `read(schema, table, row_id, column)` – If the given path is in the cache, return it. If not, using the transaction's start time (see below) and the path, retrieve the requested data from the Data Store. The data store may in turn send out requests to other nodes.
- `write(schema, table, row_id, column, value)` – Create or update the given path in the cache, with the given value. Flag the path as dirty.
- `commit()` – Write back all dirty paths to the data store.
- `abort()` – Simply delete the cache. Write back nothing.

3.3 ADDRESS TABLE

The concept of the address table was introduced in the context of this work to provide a mapping of keys to host nodes. In our system, the address table is used to map the hash of a key to the node that holds the data store for the given key. Details about the address table are given in Chapter 4.

3.4 DATA STORE

The data store uses a versioned data model. The finest granularity of data is at the cell level; Data is organized in the following hierarchy:

Schema → Table → Row ID → Version → Transaction ID → Column → Data

A row's Version comes from the Lamport timestamp [18] of the transaction that updated the data. Transaction IDs are also kept in the case that two transactions with the same timestamp attempt to update the data. The greater transaction ID is always preferred.

Data Cache: The data store has a global in-memory data cache that sits between the transaction manager and the individual replica data stores. This cache supports the following operations:

- `read(schema, table, row_id, ts, column)` – Retrieve data for the given path, with the greatest version not greater than `ts`. If the path is not present in the cache, retrieve all versions from the replica data store before responding.
- `write(schema, table, row_id, ts, txn_id, column, value)` – Create the given path. If the row doesn't exist in the data store, retrieve all version for that row from the replica store before writing. Note that these writes occur only on transaction commit, and they write through to the replica data stores as well.

Replica Data Stores: Each node in Ariadne is not actually a physical machine; it is a virtual machine. The services hosted at each virtual node will see the rest of the system as a grid (our system's *logical layer*); a global distributed transactional table maps

virtual nodes to physical machines. In this way we can (1) maintain a perfect grid, no matter how many machines we have, (2) give more powerful machines heavier loads than weaker machines by assigning more virtual nodes and (3) add or remove machines from the system without disrupting the logical organization. The grid structure allows our system to organize key-key-values with the on-line partitioning algorithm, as well as support efficient global communication.

Each node has three replica copies based on the grid replication scheme, which is described in detail below in Section 4.2.2. These are simple, file-backed stores implemented using Berkeley DB [24]. They map a path `(schema, table, row_id)` to a data structure that contains the versions and columns for the path. They support the following basic operations, which are available over the network as well as locally on each node:

- `get(schema, table, row_id)` – Get the versions for the given row path
- `put(schema, table, row_id, versions)` – Merge the given versions with any existing versions for the given row.
- `overwrite(schema, table, row_id, versions)` – Replace the versions. This is used only by the condenser background process (see below).
- `delete(schema, table, row _id)` – Delete the row. This is used only by the partitioner background process (see below).

3.5 BACKGROUND PROCESSES

There are several background processes that work to complete tasks in the system. Some of them are strictly for maintenance, others are essential to completing requests.

Session Handler: Accepts new connections and parses queries. Queries are then turned into requests which are put on an API request queue. There is only one session handler.

API Request Handler: Takes an API request, and executes the query plan in the request. This handler interfaces directly with the transaction manager, which will in turn produce Data Requests. There are several API Request Handlers.

Data Request Handler: Takes a Data Request and performs the requested operation on the given data item. Data Requests may originate for remote sources. There are several Data Request Handlers.

Gossip Handler: Global state information is gossiped throughout the system. This handler takes a message, processes it, and replays it to a random selection of nodes based on the gossip algorithm. The message types it handles include:

- Transaction starts – each time a transaction begins, register its start time locally.
- Transaction ends – each time a transaction completes, either as a commit or abort, un-register its start time
- Cache invalidation – propagated for keys that have recently changed.

Data Condenser: This process has no input or output queues. It visits each data item stored on the local node and deletes old versions of the data. The condition required to delete a version is that it (1) is not the latest version and (2) is not younger than any currently running transaction. There is only one condenser. This is similar to the garbage collector described in the work of Labrinidis and Roussopoulos in 1998 [16].

Data Partitioner: Using the on-line partitioning algorithm given below in Algorithm 4.1.3, find blocks of data to migrate and manage those migrations. See below in section 4 for details on how this works. There is only one partitioner thread.

Grid Manager: The grid manager handles the process of automatically scaling up or down the cluster. This was not implemented in this work due to time constraints; our system was designed to support the model defined in Chapter 4.

4.0 IMPLEMENTATION DETAILS

In this chapter we explore the implementation details of key elements of Ariadne. We provide here design details for key items mentioned above in Chapter 3.

4.1 ADDRESS TABLE

One of our goals is to store related keys in the same node for performance reasons. In an abstract graph, however, determining which objects to store on which nodes requires some computation. The grid-address, therefore, of each key needs to be stored by our system, so that nodes know where to look for a given key. Each node has a grid-map that associates address-ranges to nodes (and physical machines).

Almost all key-value storage systems use a hash function to determine where to store a specific key-value. This has the benefit of each machine in the distributed system being able to determine the location of a key-value in a local manner. Unfortunately, the hash function will assign key-values to nodes randomly by design, without any regard for how the key-values are connected. In the case of graph data, this means that related keys are likely to be stored in a variety of places. In fact, the larger the set of related keys is, the higher the expected number of different storage sites is. When working with the graph data, an application will thus have to work with several machines – creating overheads with consistency, availability and performance. Our system eliminates such overheads by assigning keys to machines based how they are related, and using the global address table to keep track of their locations.

As the structure of the graph changes and the organization of the grid changes, key addresses will need to change. When this occurs, the Address Table is updated. The Address

Table is kept in main memory on each virtual node in the system. It is updated via gossip as address mappings change from the partitioner process. In order for the table to fit in memory, ranges of keys – blocks – are mapped to virtual nodes.

4.1.1 Determining the Block Address of a Key

When the storage system receives a request for a key, it first takes a hash h of that key and modulates h by the size of the address table.

$$b = h \mod |AT| \tag{4.1.1}$$

The resulting value b is the *natural* block address of the key. Now, the address b is looked up in the address table and the found value is used instead for b. Note that the address table is initialized to be an identity mapping; thus, in the initial state this means the lookup does not change b.

4.1.2 Determining the Value of a Key

First, the in-memory cache is used to find the value; if a value that is current with the requested Lamport timestamp is not found, then the requested block has to be retrieved from storage.

Once the block address b of the key has been determined, the primary host address is looked up via a consistent hashing ring. The primary host always has two replicas, based on the grid pattern designed by Connor et al. This is explained in detail below.

$$\text{host} = \text{findHost}(b) \tag{4.1.2}$$

If one of the replicas lives on the local host, the value is looked up from there; otherwise, a network request is sent to the closest (based on the grid) replica host.

4.1.3 On-Line Partitioning Algorithm

In order to make the Address Table work, the On-Line Partitioning Algorithm had to be modified. Originally, the algorithm found individual keys which could be moved from one virtual node to another to reduce the number of key-key-value connections between nodes. As we have discovered that mapping individual keys lead to prohibitively high latencies, we modified the original algorithm to move blocks of keys.

Theorem. *Algorithm 4.1.1 is correct.*

Proof. Given the input of blocks, and interconnections between blocks, the algorithm has one of two choices. It can either decide to swap a pair of blocks, or leave the blocks as they are. In the latter case the result is simple; as the configuration has not changed, the number of key-key-values connected between blocks has not changed. In the other case, the algorithm determines objectively the number of key-key-value connections that cross between blocks as a result of the change. It this number is negative, only then does it proceed. Thus, in any case, the number of connections will never increase as a result of the algorithm. □

4.2 LOGICAL LAYER

Each virtual node in the logical layer will be responsible for managing a set of *source keys*; each source key has some data associated with it as in a key-value store (in our example, the user's profile and status updates) and a list of *target keys* that the source key links to in the graph. These are stored in the *key-key-value table*; in our example, this would be the table that stores the follows-list for every user whose data is hosted at that node. Each key-pair, furthermore, has a value – in our example, this value can be the *follows* or *blocked* relationship.

Also stored in the same key-key-value table is an inverse mapping of keys; for every source key, each *inverse key* is stored as a `(inverse-key, source-key, value)` tuple. Consider `alice` and `bob`. The tuple `(alice,bob,follows)` is stored in the node that hosts all of `alice`'s data; the *same tuple* is stored in the node that hosts all of `bob`'s data (in his *followers-list*). If both

Algorithm 4.1.1 On-line partitioning algorithm

Require: Local store S, key-key-value connections between keys in S and other blocks

Ensure: The number of key-key-value connections across blocks is equal to or less that the count before invoking this procedure

for Each block b in the primary local data store S **do**

Let $B = \text{connectedBlocks}(b)$

where connectedBlocks finds the list of blocks where the keys between blocks are connected by key-key-values

Let $b' = \text{maxConnected}(b, B)$

where maxConnected finds the block with most connected keys

if $b' \neq b \wedge \text{countConnections}(b, b') > t$, where countConnections counts the key-key-value connections between blocks, and t is the threshold for migration. **then**

Initiate a block exchange with $H = \text{host}(b')$

The other host H must pick a (potentially empty) block b^* to exchange.

This is done using the same counting strategy.

Versions from all replicas of both blocks are merged.

The non-primary replicas become read-only.

An update to the address table is then gossiped through the system.

At the same time, the blocks are copied between the hosts.

During the copy, the hosts communicate with one another to answer queries about the blocks.

Updates are kept on the (new) primary replica.

Updates are merged into the block once the exchange is complete.

end if

end for

`alice` and `bob` are hosted in the same node, then *only one* `(alice,bob,follows)` *tuple needs to be stored* in that node.

As stated earlier, virtual nodes in our system are mapped to a grid. This grid maintains the logical organization of the network – it is the basis for decisions about where replicas are placed, and where messages are sent to and pulled from. The grid-map is a global distributed structure that stores the `(row-range, column-range, physical-machine)` relation – the *row-range* and *column-range* give the ranges of key addresses; *physical-machine* gives the physical network address of the machine that hosts the virtual node that manages keys with addresses that fall in the given ranges.

The grid-map furthermore supports transactions; this is because some updates to the grid-map involve changing many parts of the map atomically and the grid-map must be consistent for all nodes. Other updates involve changing only one cell of the grid, as we will see later. We posit that during normal systems operations, changes to the grid-map will be infrequent and applications will need information from a limited number of locations in the grid-map in a session – a side-effect of locality by computation (discussed below in Section 4.2.1).

In order to support such transactional operations, most cloud data stores have some unified protocol for communication between nodes. These can cover both sending messages about system state [8] and propogating replicas [11]. Our system will de-couple these; it will propogate replicas passively and use a distributed mutual exclusion protocol (DME) similar to Maekawa's [22] for system state updates.

Our distributed mutual exclusion scheme (Fig. 4.2.1), like Maekawa's, use a finite project plane (or *FPP*). An FPP is a geometric structure characterized by the following axioms: *every point is incident on exactly two lines* and *every line passes through exactly two points*. This is at the core of Maekawa's protocol – a node in the distributed system corresponds to one point and one line in the FPP. This technique requires between $3\sqrt{N}$ and $5\sqrt{N}$ messages to create mutual exclusion – an asymptotic improvement over the previous methods that required $\Theta(N)$ messages [22].

The key-key-value system will use a relaxed version of FPPs in its communication protocols. As stated above, each physical machine in our system can have several virtual nodes,

which are organized in a grid. Each virtual node will send system state updates to and request state information from nodes that lie in the same row and column. This ensures the following properties: every point is incident on *at least* two lines and every line passes through *at least* two points. Think of each virtual node representing a point and a line. The *point* that the node corresponds to is its grid cell. The *line* that it corresponds to is the set of virtual nodes that lie on the same row and column. Suppose some node A sends a message to every point-node incident on the A-line. If another node B requests messages from every point-node on its B-line, then the message from A will reach B through *two* point nodes (see Figure 4.2.1). In this way, if a failure in one of the connecting nodes occurs, the message will still be passed.

Load Maintenance Operations: Each logical node has a set of operations available to it for completing application requests and for managing load. The load management operations allow a virtual node to split if it is overloaded, and merge with neighbor virtual nodes if all nodes are underloaded. When these operations are performed, they must maintain the structure of the grid (as in Figure 4.2.2), and for this reason distributed mutual exclusion is necessary to ensure that a history's splits and merges are serializable for the grid. For instance, if some node X receives two separate requests to split, then it can accept both requests without compromising the grid structure. On the other hand, if X receives a request to split row-wise and merge column-wise, it must choose (based on the logical clock of requests) which request to honor.

Splitting: Splitting is necessary to prevent overloading; when a node C has load greater than some threshold, it initiates a split. The first step is to contact each node X that lies in the same row or column as C (referred to as the line l, see Figure 4.2.2) and determine if a split is possible and to acquire the split-lock. If there is another node Y in l that is involved in a conflicting merge operation, Y contacts the merging node Z to make sure that the merge will succeed. If the merge will succeed, then C must wait to split; otherwise, C tells all nodes in l to split. Once each splitting node has succeeded, then C updates the grid map and releases its split-lock.

Merging: When some virtual node C has load less than a certain threshold (measured by CPU utilization and disk utilization) it contacts each adjacent node C_j to test if a merge

is possible (see Figure 4.2.2). If each of three C_j responds that it has load less than the threshold, each C_j contacts every node X_k that lies in the same row and column to determine X_k's load and propose a merge. If every X_k that was contacted has load below the threshold, then C tries to acquire the merge-lock. For each set of merging nodes $C_1, ..., C_4$ a node C is created that handles keys in the in unioned key-range of nodes $C_1, ..., C_4$. Each of the nodes $C_1, ..., C_4$ stays on-line, but only as a router that forwards every request to C. All nodes X_1, X_2 in the merging rows and columns similarly merge into new nodes X, as illustrated in Figure 4.2.2. Once C has confirmed that the merge has succeeded, it updates the grid-map and the nodes $C_1, ..., C_4$ are notified by C that they can shut down; finally, the merge-lock is released.

In the event that a failure is detected in either process (by timeout), each node that becomes aware of the failure considers the operation cancelled. The current failed operation is reversed and the relevant locks released. Once recovery has occurred and has been confirmed, the originating node C will re-start the operation if necessary.

4.2.1 Computation of Locality

Although a key's grid-address is assigned arbitrarily when it is first inserted into a virtual node, the node that it first resides in may not be the optimal place to store that key. Our system, thus, needs to be able to determine reasonably good locations for keys, such that related keys are stored in the same or grid-adjacent virtual nodes. In order to achieve this, we introduce the concept of *computation of locality*.

Spatial data structures, especially grid files, take advantage of spatial locality by storing items that are (spatially) related to one another in the same place, or at worst in a nearby place. Note that traditional grid files achieve this locality by hashing keys to rows and columns in the grid; a cell in the grid represents a set of key-pairs determined by the hash function [23]. For our system, which stores graph data, this scheme will cause related keys to be spread out across whole rows and columns. Instead, our key-key-value system will store all data for some deliberately selected, related keys in one grid cell.

Unfortunately, for abstract structures such as social networks, spatial locality is not intrinsically available for systems to exploit. We propose to use a novel on-line graph partitioning algorithm to determine the grid-addresses of keys. On each node, a continuously-running background process calls the on-line partitioning algorithm (Algorithm 4.1.3) on each block of keys in an arbitrary order. The structure of the graph changes in the following events, which may lead to the migration of source-keys:

Split Event: When a virtual node splits, each of the source-keys stored in that node must be re-addressed. The node arbitrarily assigns a new grid-address to each key. The keys are then immediately transferred and the Address Table is updated; in this way the split can be completed quickly and the new nodes can begin operations. Once the split has succeeded, the new nodes start up the on-line partitioning algorithm to better organize the recently moved keys.

Merge Event: When a group of nodes merge, the process is much simpler: each source-key is updated with the new node's grid-location intervals. Merging has the benefit of reducing storage overhead in the new node by eliminating duplicate key-key-value entries in the merged nodes. Also, all of the edges that crossed between the merged nodes will be contained in the same node; improving both performance and availability.

Target Key Insert or Delete Event: When a user inserts or deletes a key-pair, a target key is added to or removed from a source key's list. A potentially better arrangement of the source-keys may then exist. If the source key needs to be moved, a copy is made on the new hosting node. The address table is then updated with the new location; finally, the old host node will delete the source-key from its store.

Within each node's data store, we maintain not only the key-key-values and source-key's data, we also maintain some aggregates about each source-key. These include a list of nodes that host the source-key's target-keys and inverse-keys, and the number of target-keys and inverse-keys hosted on each node. Using these counts, our on-line algorithm can quickly determine if a key can be placed in a better location. After each key-pair insertion or deletion, these counts are updated as part of the operation. Our proposed algorithm is similar to that proposed by Zanghi et al [33]; however, their work focused on the on-line addition and removal of graph vertices whereas we are concerned with operations on edges.

To our knowledge, the only other similar approaches to ours are those by Sun et al [30], Kernighan and Lin [15], and Fiduccia and Mattheyses [12]; these work in an off-line fashion.

4.2.2 Replication

Each node has a set of key-ranges that it is responsible for replicating. These ranges are defined as the set of keys in the same logical column *or* the same logical row as the node. Suppose that node A is sending copies of data item d to d's replicating nodes. A first sends it to the adjacent nodes in the grid; each of which returns a success or failure message. A counts the number of distinct physical locations that have received a copy, and if that number is less than some threshold W, A sends the update of d to the next adjacent nodes in its row and column. This is repeated until W or more physical copies have been made. Our system, thus, uses $W + R \leq N$ replication, in the interest of greater availability and lower response times [32].

Conflict Resolution: Suppose that some node Z receives two or more conflicting updates d_1 and d_2 of a `(source-key,target-key)`. Each of d_1 and d_2 has a set of versions identified by Lamport timestamps [18]; each record contains a Lamport timestamp and a randomly-generated transaction ID. Z will merge the versions of d_1 and d_2 – in a case where two records have the same Lamport timestamp, the transaction ID is used to determine the order of versions.

4.2.3 Physical Layer

In any large-scale system, the ability to add and remove hardware resources as demands change and recover gracefully from hardware failures is essential [11]. We subdivide these hardware changes into two categories: *planned* and *unplanned*. Planned changes include incorporating a new machine, taking an old machine off-line or temporarily removing a machine for maintenance and upgrades. Unplanned changes are the result of unexpected hardware or software failures; generally, these lead to a machine becoming isolated from the network – our system utilizes existing well-known distributed and cloud recovery schemes [11, 25, 6] to handle physical failures.

Adding Resources: Adding resources generally means adding a machine to the network. When this is done, the *physical controller* running on the machine starts up and takes over a virtual node from another physical machine. Occasionally, a logical split needs to take place before the migration is complete. Once the migration of a virtual node from one machine to another is complete, the grid-map is updated to reflect the node's new location.

Removing Resources: When a machine needs to be removed from the system, an administrator connects to that machine's physical controller and issues a shutdown command. The machine then tries to migrate all of its virtual nodes to other machines. When this succeeds, it shuts itself down. In some circumstances, this can be done automatically – e.g., the machine senses that it is over-heating or detects disk failures.

Handling Failures: In the event that a machine fails or becomes isolated from the system suddenly and unexpectedly, the virtual nodes that this machine maintained need to be re-created elsewhere. When a failure is first noticed in some virtual node F, the node that has detected the failure (by timeout) will try to become the *recovery coordinator* for F using the Fast Paxos algorithm [19]. The recovery coordinator node R will contact F's logical row and column-adjacent neighbors to see if they too have failed. If none of the immediately adjacent nodes F_j are alive, then R contacts each F_j's adjacent row and column-neighbors to see if they have failed. R repeats this expansion until it has discovered the full set of adjacent failed nodes. It then divides the set up, and contacts each of the active nodes adjacent to the failed region with a subset of failed nodes that need to be taken over. The physical controller for each contacted machine then creates the virtual nodes – each virtual node then uses the `pull` operation to get its replicas and update the grid-map. Each of the other potential recovery coordinators will attempt to become the coordinator at a specific interval. Should the current recovery coordinator fail, this guarantees that another will emerge.

The default ordering-consistency is guaranteed even in the case of such failures. Recall again the example where `bob` blocks `eve`, and then makes a status update. Suppose now that after receiving the *block* update from some replica node X, node F fails before it receives the status update. In this case, when F is recovered, one of three things will happen: (1) F recovers from some node that has received neither the *block* update or the status update, (2) F recovers from some node that has received the *block* update, but not the status update –

or (3) F recovers from some node that has recieved both updates. In any of these scenarios, the private status update is not propogated to F before the *block* update.

4.3 DATA MODEL IMPLEMENTATION

Our system uses key-key-values in computing joins. At this point, we need to introduce the implementation details of the data model. Although from a user's perspective Ariadne allows the creation of any number of tables, there are in reality only three system tables used for managing the data. These are:

- **key-values** – the mapping of primary keys and relation names to data
- **key-key-values** – the mapping of primary keys and relation names to child relation names and primary keys
- **schema** – the mapping of column names and relation names to parent relation names

Key-values stores raw user data. It is a relation of the following structure, where a relation name and ID identifies a relation:

$$(\underline{\text{Relation}}, \underline{\text{ID}}, \underline{\text{Column}}, \text{Value}) \tag{4.3.1}$$

Key-key-values stores the connections between tuples in **key-values**. Specifically, it lists all of the tuples that have a foreign key to a specific parent tuple. It is defined as the following relation:

$$(\underline{\text{Parent Relation}}, \underline{\text{Parent ID}}, \underline{\text{Child Relation}}, \underline{\text{Child ID}}, \text{State}) \tag{4.3.2}$$

where State is simple a flag indicating that a connection exists. There are a couple of subtleties here worth mentioning. The first is that the mapping for some primary key k in some parent table P, a special table named as $P.k$ is created. For any child table C and primary key l in C, a column $C.l$ is created in the table $P.k$. In this way, all of the mapping data is (1) guaranteed to be stored on the same block and (2) a full list of child tables and keys can be retrieved by asking for all of the columns of $P.k$.

Note that we do not need to explicitly store child to parent connections; these are stored for us by **key-values**. The foreign key column in the child relation, specifed earlier by the `CREATE LINK` command, is known to store the primary key of the child relation for that row.

Schema stores the network of connections between user relations. It is the following relation:

$$(\underline{\text{Child Relation}}, \underline{\text{Child Column}}, \text{Parent Relation}) \tag{4.3.3}$$

Note that the parent to child relationships do not need to be explicitly stored, as they are managed in **key-key-values**. The **schema** relation is directly updated by `CREATE LINK` and `DELETE LINK` queries.

4.4 `UPDATE` IMPLEMENTATION

`UPDATE` queries are restricted in that a query can modify only one tuple. This tuple is identified by the `WHERE` clause, which must give an equality condition on the primary key (the primary key is always `id`). The update query must provide a list of columns to update. For example,

```
UPDATE profiles SET first_name='Jane', last_name='Eyre' WHERE
id='RU239e9RNT';
```

This query will set the first name and last name columns of the `profiles` relation for the tuple with primary key (`id`) `RU239e9RNT`.

For this example, the system will simply find the tuple identified, and if it exists, update the given fields. If it does not exist, it will be created, with only the two given fields set. Consider this example:

```
UPDATE photos SET data='arTN92nTE0...', profile_id='RU239e9RNT' where
id='INWU02en20';
```

Let us assume that this query was run at some point in the past:

```
CREATE LINK photos.profile_id TO profiles;
```

This means that there is a foreign key relationship between `photos` and `profiles`. As such, the system must not only update the tuple for `photos` in the system table **key-values**, it must update **key-key-values** as well to reflect the foreign key link. It will need to access the **schema** beforehand to know which column is the foreign key, and what relation it references. Because of this, every update follows the same three-step plan:

1. Look up the given user relation and columns in **schema**, to find a list L of foreign relations
2. For each foreign relation found, if the value in the update is not `NULL`, then set the link in **key-key-values** between the given primary key and the foreign key.

   If the value is `NULL`, destroy the link in **key-key-values**
3. Find the identified tuple and update the given values.

4.5 KEY-KEY-VALUE JOIN ALGORITHM

Our join algorithm is essentially hash join; the critical distinction here is that it leverages the **key-key-values** [7] – using that relation as a clustered hash map.

Theorem. *Algorithm 4.5.1 is correct.*

Proof. There are two cases to prove, regardless of which relation is larger. Note that this choice is entirely one of efficiency, it does not affect the correctness of the algorithm. *Case 1:* $r_s = r_1$. In this case, we find rows in r_2 by the foreign key in r_1. Each constructed row thus meets the three criteria because (1) Its left part is from a row in r_1, (2) its right part is from r_2 and (3) the right part originated from the equality search by id in r_2 for the foreign key r2_id from r_1. *Case 2:* Here we assume that *kkv* correctly matches parent rows to child rows based on the input. Thus, for each row r in r_2, we know that for each row $r' \in r_1$ that $r'_{\text{r2\_id}} = r_{\text{id}}$. The left and right parts come from r_1 and r_2 respectively, satisfying our conditions. □

As all relations are hash-indexed on the primary key **id**, the algorithm works in the same way as hash join [34] in Case 1. Case 2, however, makes our algorithm unique. Suppose

Algorithm 4.5.1 Key-key-value join algorithm

Require: Two relations $r_1(\underline{\text{id}}, \text{r2\_id}, ...)$ and $r_2(\underline{\text{id}}, ...)$, key-key-value relation $kkv(\underline{\text{Parent Relation}}, \underline{\text{Parent ID}}, \underline{\text{Child Relation}}, \text{Child ID})$. Note that any of r_1, r_2 may be temporary relations generated by another join as part of the query plan. As such, we also take a source relation mapping s which maps every relation to a base relation in the system, that could have entries in kkv. Note that s is the identity mapping for base relations.

Ensure: Relation $r_3(\underline{\text{r1.id}}, \text{r2.id}, ...)$ such for each row $r \in r_3$, $\pi_{r_1.*}(r) \in r_1$, $\pi_{r_2.*}(r) \in r_2$ and $\pi_{r_1.\text{r2\_id}}(r) = \pi_{r_2.\text{id}}$. We also output the updated source relation mapping s with $s(r_3)$ mapped.

Let r_l be the larger of r_1 and r_2.

Let r_s be the smaller. For each row $r \in r_s$, apply the following.

if $r_s = r_1$ **then**

Let $r' = \sigma_{\text{id}=k}(r_l)$

where $k = \pi_{\text{r2\_id}}(r_s)$.

Let $(r + +r') \in r_3$.

else if $r_s = r_2$ **then**

let $C = \sigma_{\text{Parent}=s(r_2),\text{Parent ID}=k,\text{Child}=r_1}(kkv)$,

where $k = \pi_{\text{id}}(r_2)$.

for each $k' \in C$ **do**

let $r' = \sigma_{\text{id}=k'}(r_1)$.

Let $(r + +r') \in r_3$.

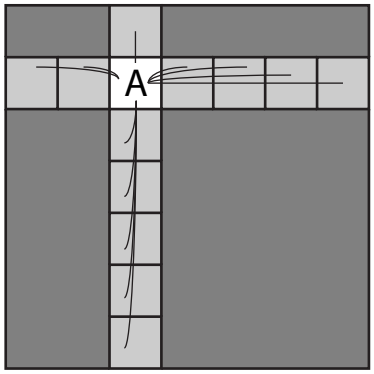
Let $s(r_3) = s(r_1)$.

end for

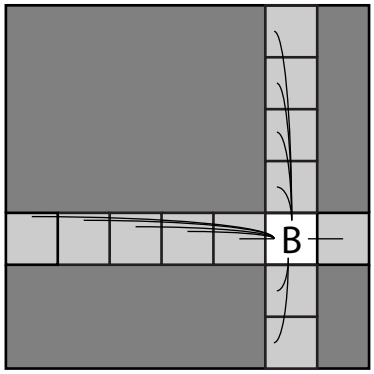
end if

that r_2 has a single tuple, and r_1 has millions of tuples. Most likely, to scan all of r_1 for foreign keys that point to the single record in r_2 would be terribly inefficient. Now, the power of key-key-values comes into play – because our system has diligently kept track of the connections between records, we can simply take the key of the single record in r_2 and map it to a list of keys in r_1, dramatically reducing the cost of our query.

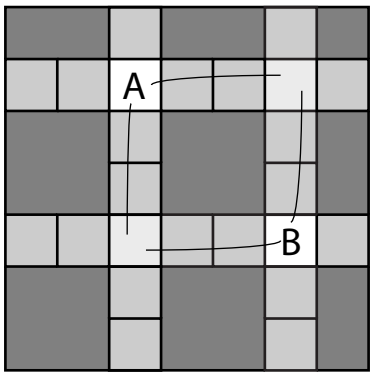
Furthermore, because our system partitions keys based on their relationships, all of the related records in r_1 are likely already available on the same host as the record in r_2. This means that network latency will have a minimal impact on the query's response time.



A sends to all nodes on its line.



A's message reaches B through two nodes.



B reads from all nodes on its line.

Figure 4.2.1: FPP-based communication in the grid.

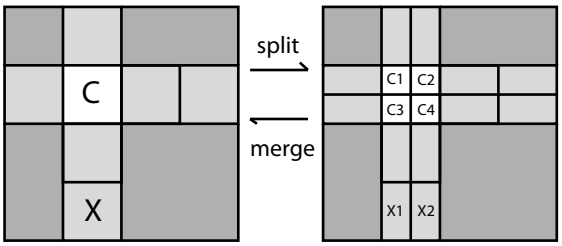


Figure 4.2.2: Splitting and merging a node.

5.0 EVALUATION

5.1 EXPERIMENT SETUP

Experiments were run on clusters of Amazon ElasticCloud 2 virtual machines. Common traits of these machines are given in Table 5.1.1 In our experiments, we compare MySQL

Table 5.1.1: Experimental machine specifications

CPU Speed	2.33 GHz
No. CPU cores	1
Main Memory Size	1.7 GB

Cluster with our proposed system Ariadne. As these systems run on different underlying technology, we provide the specifications here:

Table 5.1.2: Ariadne specifications

JDK Version	1.7
JDK Provider	Oracle

5.1.1 Experimental Workload

Our workload was designed to simulate the variety of requests a social network back-end would produce under a variety of conditions. The workload also explores extremes, so that we may reason about how our system behaves under such circumstances as well. The test

Table 5.1.3: MySQL Cluster specifications

MySQL Server version	5.5.22
NDB Server version	7.2.6

Table 5.1.4: Test database schema

id	primary key
data	
tN_id	foreign key to table N (not present for first table in schema)

schema (see Table 5.1.4) we used included up to four tables; each table (except the first) has a primary key column, a data column, and a foreign key column to the next table. In the MySQL version of this schema, the foreign key columns and primary key columns were indexed, and foreign key constraints were created.

Table 5.1.5: Experimental workload parameters

No. Writes / No. Queries	.1 to .9
Concurrent Threads (per node)	10 to 1000
Nodes	4
Select Type	Single Table, Join
Primary Key Space	10 to 1000

5.2 PERFORMANCE ANALYSIS

We have evaluated our system, allowing us to support our claims of performance and eventual consistency. These experiments focus on overall system performance, to expose trends in response time as other factors change. As our system has not yet had the thorough optimization that MySQL Cluster has had, we naturally expect that in absolute terms MySQL will outperform Ariadne. We want to point out, however, patterns in performance between these systems that suggest Ariadne is more scalable and versatile than MySQL.

5.2.1 Write Mix

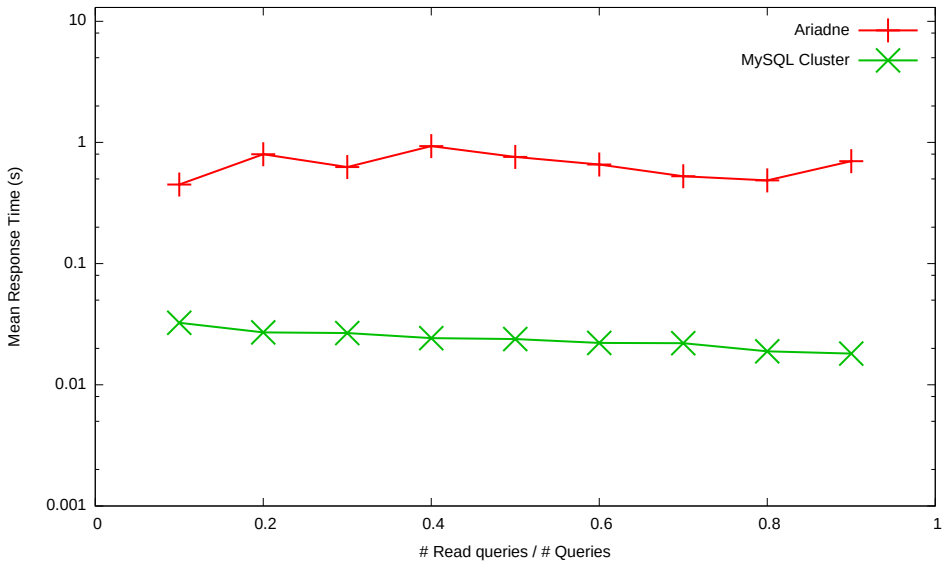


Figure 5.2.1: Effect of varying read/write mix on response times for Ariadne and MySQL. Although Ariadne's response times are higher than MySQL's, Ariadne's are uniform across different mixes, indicating high write availability as well as high read availability.

In Figure 5.2.1 we examine the effect of changing workload mixes on response times. Both lines appear mostly flat – if one looks carefully however, one will notice a clear trend favoring reads in MySQL. One of Ariadne's design goals is to be equally available for writes as for reads, because almost all workloads from web applications can be heavy at times.

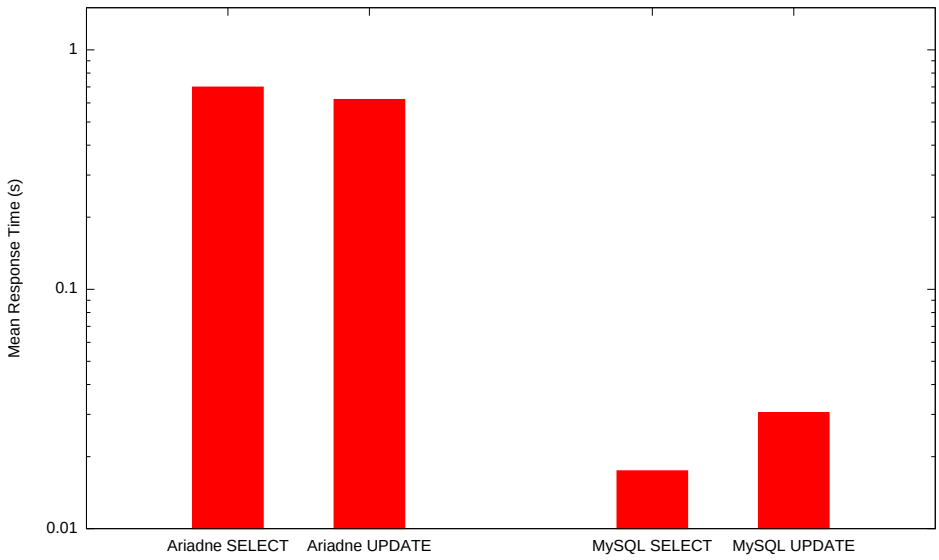


Figure 5.2.2: Effect of different query types on response time, as the number of concurrent threads ranges from 10 to 562. Response times for Ariadne indicate that Ariadne is equally available for writes and reads.

In Figure 5.2.2 we see response times for different query types. This result very clearly shows that MySQL reads are faster than writes. Ariadne, however, provides a near equal response time for both types. Although these response times are absolutely higher, Ariadne can as part of our future work be further optimized to provide better overall performance, while maintaining the balance of performance between reads and writes.

5.2.2 Query Type

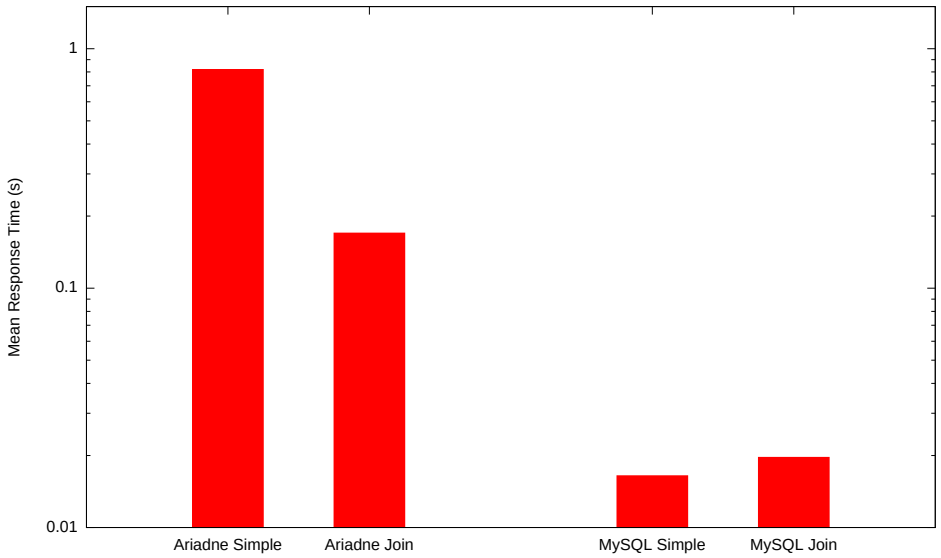


Figure 5.2.3: Effect of different select types on response time. Response times for Ariadne indicate that Ariadne is optimized for join queries.

In Figure 5.2.3 we see response times of read queries only. These are subdivided into simple queries, which reference one table only and join queries, which reference two joined tables. As above, the join is always an equality join over a foreign key. In the result, we see that for MySQL, the join is more costly than the simple query. Ariadne's result, however, is starkly different – the join is faster than the simple query. This is because Ariadne's superior ability to leverage locality of reference in data.

5.2.3 Concurrent Threads

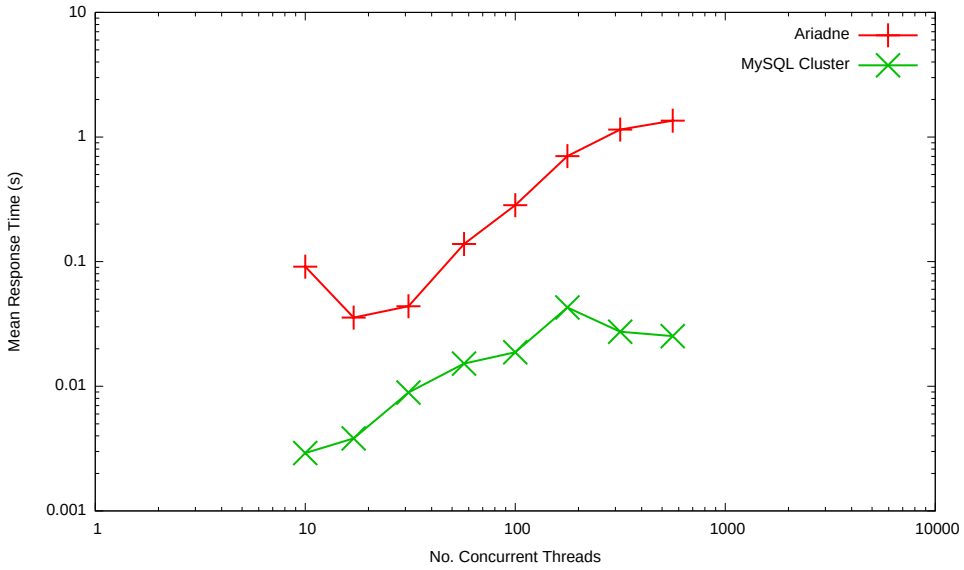


Figure 5.2.4: Comparison of response times between MySQL and Ariadne, as the number of concurrent threads ranges from 10 to 562. Although Ariadne has higher response times, the profile follows that of MySQL.

In Figure 5.2.4 we see responses times as the number of concurrent threads varies. One will notice that aside from the spike in response time at 10 threads, response times increase at a linear rate. This is also true of MySQL, again with one exception just above 100 threads. These spikes can be accounted for by artifacts of the test environments.

5.3 SENSITIVITY ANALYSIS

5.3.1 Write Mix

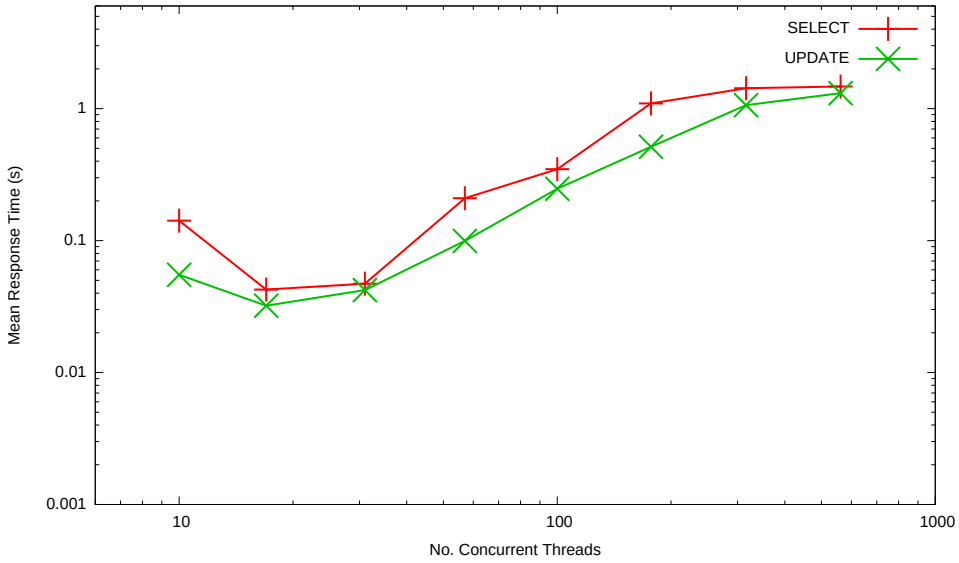


Figure 5.3.1: Effect of different query types on response time. Ariadne is equally available for writes and reads.

In this section, we examine the effect of varying the mix of read and write queries in the workload. In Figure 5.3.1, we note that mean response times increase as load on the system increases. We consider this to be normal behavior. One should note, though, that the average response times for reads is close to that of writes at each data point. This tells the reader that he or she can expect the same level of availability for reads as well as writes. We identify this as a key quality of high-availability data stores.

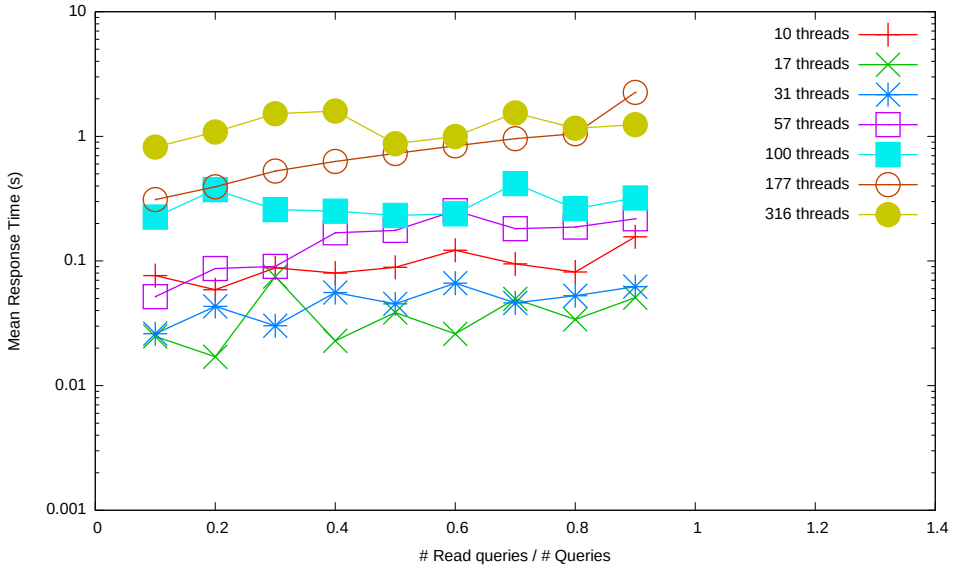


Figure 5.3.2: Effect of varying read/write mix on response times for Ariadne. Response times are uniform across different mixes, indicating high write availability as well as high read availability.

In Figure 5.3.2, we evaluate the effect of different read/write mixes on overall system response times. Mean response times are taken over all queries in the mix, so as to measure total system performance. The results are broken down by the number of concurrent threads per node. One should note that response times consistently increase with the number of threads. Most importantly, response times are uniform across the different mixes. This tells us that Ariadne performs well for all workload mixes.

5.3.2 Select Type

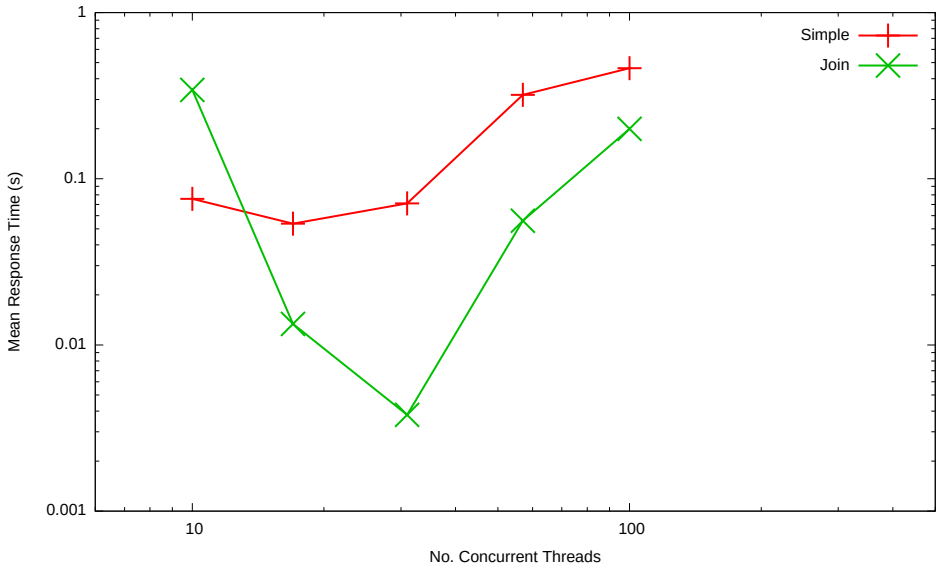


Figure 5.3.3: Effect of different select types on response time, as the number of concurrent threads ranges from 10 to 100. Response times for Ariadne indicate that Ariadne is optimized for join queries.

In Figure 5.3.3, we examine the mean response time of simple select queries and join queries. The join queries join two tables using an equality join between the primary key of the parent table and the foreign key in the child table. One may observe that response times dip and spike for joins, while they steadily increase for simple queries. This is because the amount of load in the dip is such that it is high enough to keep the cache warmed (which cannot fully happen with lower loads), yet not so high that writes cause frequent invalidations. The nature of joins require that more data be pulled than simple selects, allowing the cache to be better populated than it would be with simple queries at the same level of load. These results show us that Ariadne is able to efficiently support joins, as well as simple queries.

5.3.3 Primary Key Space

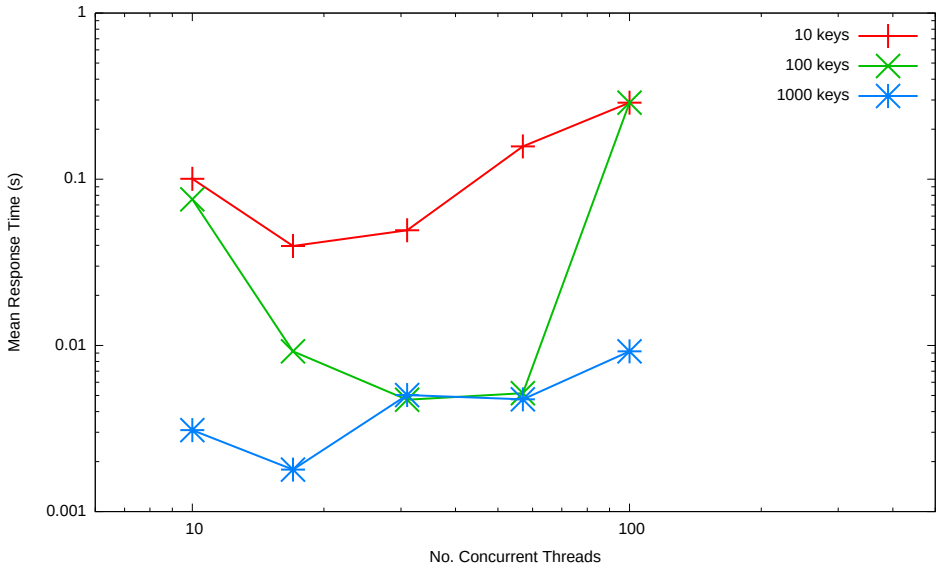


Figure 5.3.4: Effect of varying primary key space on response times for Ariadne, as the number of concurrent threads ranges from 10 to 100. Response times are higher for smaller key space, due to cache misses caused by frequent writes across the small space.

In Figure 5.3.4, we examine the impact that varying the primary key space has on response times. One will observe three different trends for the different spaces. In the 1000-key space, we see the best performance. This is because the space is small enough that all of the data can be cached in memory, while large enough that writes do not cause a high frequency of cache invalidation for individual keys. The 10-key space has the worst performance. This is because the space is so small that updates cause a high frequency of cache invalidations, forcing queries to go to I/O on each retrieval.

The 100-key space shows an interesting pattern. Extremely low numbers of concurren threads and extremely high numbers of threads result in response times similar to the 10-key results. Non-extreme thread counts, however, give results similar to the 1000-key results. In the low extreme this is due to cache misses that arise because the key-space is larger than the number of threads. In the high extreme, the spike in response time arises because of cache misses due to frequent invalidations.

5.4 ON-LINE PARTITIONING ALGORITHM

In addition to evaluating our overal system in terms of the number of concurrent threads and types of workload, presented earlier, we also briefly present the experimental evaluation of our proposed on-line partitioning algorithm. So as to evaluate our proposed algorithm, we have generated graphs with incoming branching factors between 10% and 100% of the number of vertices in the graph. Branching factors fall on a Zipf distribution with $\alpha = 1.5$. This is to reflect the observation that in social networks, some users have far more followers than others.

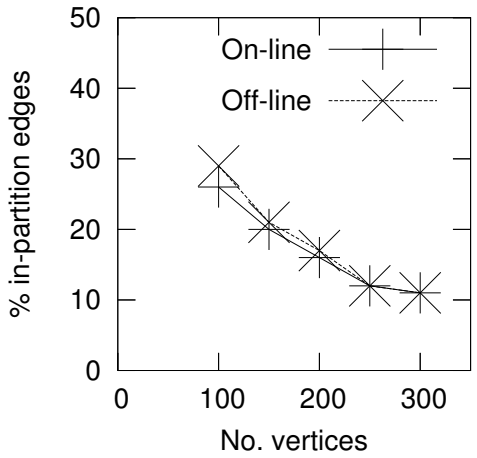


Figure 5.4.1: Experimental evaluation of the proposed on-line algorithm's effectiveness. On-line algorithm performs as well as off-line.

In our experiments, we generated three graphs for each evaluated graph size (number of vertices) and fixed the partition size to be at most 30 vertices. We measured the percentage of edges that cross between partitions and the number of vertices moved between partitions for both our proposed algorithm and the off-line algorithm from the work of [12] and [15]. In our implementation of the off-line algorithm, the graph is partitioned hierarchically until all partitions are smaller than the size limit. In each partitioning, the algorithm is allowed to iterate until the increase of the moving average of the improvement from each of the last ten iterations is greater that 99%. For our on-line algorithm, we generate a random sequence of

edge insertions and run the algorithm periodically, after every $|V|/10$ insertions where $|V|$ is the number of vertices in the graph.

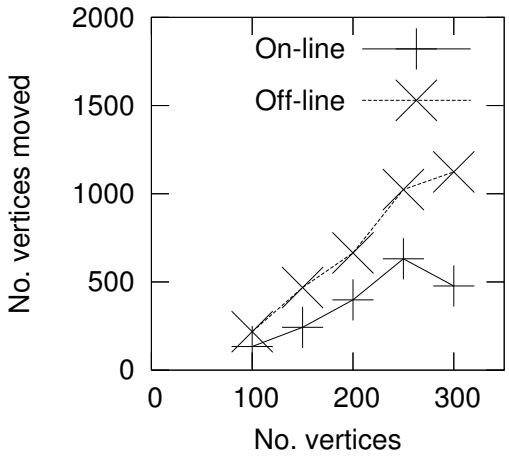


Figure 5.4.2: Evaluation of the proposed on-line algorithm's performance; on-line performs better than off-line

In Figure 5.4.1 we see that our proposed on-line algorithm does about as well in terms of partitioning as the off-line algorithm, with the benefit of higher speed and parallelizability. This improvement in performance is apparent in Figure 5.4.2; where the average number of moved vertices is far less for the on-line algorithm than for the off-line algorithm.

6.0 CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS

In this work we have examined a novel data store named Ariadne. Ariadne is unique among data management systems as it balances decentralization, performance and high availability with transactional guarantees and a simplified SQL interface. Its subsystems use the principles of key-key-value stores to optimize data locality for high performance. We have, thus, contributed the following:

1. A simple SQL-like interface that supports basic joins
2. Transactional guarantees similar to snapshot isolation
3. A near relational model
4. A novel model for scalable data stores that extends the key-value model, the *key-key-value* store
5. A high-level design for the *key-key-value* system
6. An implementation of the Key-Key-Value store design

Through empirical evaluation we have demonstrated its performance characteristics and versatility across various workloads. Although we have established above that is there still work to be done in the realm of performance optimization and empirical analysis, we have through our analysis in this work established the foundations for future work in this new space of key-key-value based systems.

6.2 FUTURE WORK

Although the work we have completed indeed shows promise for Ariadne, there are several areas we feel worthy of further investigation. These include:

- Overall System Optimization – Although Ariadne implements several cutting-edge technologies, its performance is hampered due to problems with scheduling and request management. The system will need to be optimized to more intelligently prioritize request threads, back-end service threads and sub-system threads. This and other code optimization will bring Ariadne's performance ahead of similar production systems.
- Analysis of Partitioning Algorithms – Our system is an implementation of the key-key-value store paradigm [7]. There is much to be explored in this space, including new partitioning algorithms and techniques for managing data locality.
- Further Scalability Analysis – Future work should also investigate Ariadne's scalability, especially with large installations. Unfortunately, due to budgetary constraints, we were not able to explore this in the current work. We are confident, however, that an in-depth exploration will validate our claims of scalability. Furthermore, future work should also explore various algorithms for expanding and contracting Ariadne clusters.

In the interest of facilitating future work, source code for Ariadne is freely available at https://bitbucket.org/agconnor/ariadne.

Our work is an early step in what we hope will be a series of explorations into systems that challenge the foundations of "NoSQL" systems. We hope that through our work, the community will see the continued value of query languages, transactional guarantees, and join support for scalable, high-availability systems.

BIBLIOGRAPHY

[1] http://www.facebook.com/note.php?note_id=24413138919.

[2] Project voldemort. http://www.project-voldemort.com/.

[3] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, August 2009.

[4] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide Time to Relax*. O'Reilly Media, Inc., 1st edition, 2010.

[5] 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 *Proc. 7th USENIX Symp. on Oper. Syst. Design and Impl.*, pages 205–218, 2006.

[6] Manhoi Choy and Ambuj K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. In *Proc. ACM symp. on Theory of computing*, pages 593–602, 1992.

[7] A.G. Connor, P.K. Chrysanthis, and A. Labrinidis. Key-key-value stores for efficiently processing graph data in the cloud. In *Proceedings of the 2nd International Workshop on Graph Data Management held in conjunction with the 2011 IEEE 27th International Conference on Data Engineering*, pages 88 –93, April 2011.

[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. In *Proc. VLDB Endow.*, volume 1, pages 1277–1288, 2008.

[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, , and Dale Woodford.

Spanner: Google's globally-distributed database. In *Proceedings of OSDI'12: Tenth Symposium on Operating System Design and Implementation*, pages on–line, 2012.

[10] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proc. 1st ACM symposium on Cloud computing*, pages 163–174, 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. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[12] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. 19th Design Automation Conference*, pages 175–181, 1982.

[13] Michael J. Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '82, pages 70–75, New York, NY, USA, 1982. ACM.

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

[15] B. W. Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49:291–307, 1970.

[16] Alexandros Labrinidis and Nick Roussopoulos. A performance evaluation of online warehouse update algorithms. *Department of Computer Science, University of Maryland, Technical Report CS-TR-3954*, pages pp. 1–26, November 1998.

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

[18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[19] Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, 2006.

[20] Edmond Lau and Samuel Madden. An integrated approach to recovery and high availability in an updatable, distributed data warehouse. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 703–714. VLDB Endowment, 2006.

[21] Kevin Loney. *Oracle Database 11g The Complete Reference*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2009.

[22] Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Computer Syst.*, 3(2):145–159, 1985.

[23] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.

[24] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[25] "M. T." Ozsu and P. Valduriez. Distributed database systems: where are we now? *Computer*, 24(8):68 –78, August 1991.

[26] Eelco Plugge, Tim Hawkins, and Peter Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, Berkely, CA, USA, 1st edition, 2010.

[27] M. Ronstrom. On-line schema update for a telecom database. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 329 –338, 2000.

[28] Mikael Ronström and Jonas Oreland. Recovery principles of mysql cluster 5.1. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 1108–1115. VLDB Endowment, 2005.

[29] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littleeld, and Phoenix Tong. F1 - the fault-tolerant distributed rdbms supporting google's ad business. In *SIGMOD*, 2012. Talk given at SIGMOD 2012.

[30] Yizhou Sun, Jiawei Han, Peixiang Zhao, Zhijun Yin, Hong Cheng, and Tianyi Wu. Rankclus: integrating clustering with ranking for heterogeneous information network analysis. In *Proc. EDBT: Advances in Database Technology*, pages 565–576, 2009.

[31] Hoang Tam Vo, Chun Chen, and Beng Chin Ooi. Towards elastic transactional cloud storage with range query support. In *Proc. VLDB Endow.*, volume 3, pages 506–517, 2010.

[32] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[33] Hugo Zanghi, Christophe Ambroise, and Vincent Miele. Fast online graph clustering via erdos-rnyi mixture. *Pattern Recognition*, 41(12):3592 – 3599, 2008.

[34] Hansjörg Zeller and Jim Gray. An adaptive hash join algorithm for multiuser environments. In *Proceedings of the 16th VLDB conference*, pages 186–197.