LEVERAGING METADATA IN NOSQL STORAGE SYSTEMS

BY

ALA WALID SHAFE ALKHALDI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Advisor:

Associate Professor Indranil Gupta

ABSTRACT

We present Wasef, a metadata system for NoSQL data stores. Wasef allows us to support important features such as data provenance, flexible data operations, and new administration tools. Our system has a general design that can be made compatible with most NoSQL data stores without imposing heavy performance overhead.

We implement a customized version of the Cassandra data store augmented with Wasef, and we compare it experimentally with the standard version. We also provide three use-case scenarios that exploit the metadata collected by our system. First, we support a flexible mechanism for dropping columns that addresses an open major issue in Cassandra 1.2. Second, we make provenance information for Cassandra queries available to the data store clients. Third, we provide a tool for verifying node decommissioning from Cassandra cluster. Our evaluation shows that Wasef provides linear scalability, low throughput overhead of 9%, and read latency overhead of less than 3% compared to the non-metadata default Cassandra. In addition, our system provides steady read and update latencies regardless of the metadata size.

I would like to dedicate my thesis to the Fulbright Scholar Program for giving me the opportunity to pursue my master degree at The University of Illinois at Urbana-Champaign, and to my parents and wife for their love and support.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

NoSQL data stores are growing in their usage. The introduction of Bigtable [1] and DynamoDB [2] opened the door for many close and open source variants including Cassandra [3], Voldemort [4], and MongoDB [5]. Although these systems vary in the implementation details, they share many design aspects such as simple data model, minimal set of supported operations, weak security model, scalability, and high availability.

The data model is generally formed by a group of tables, also known as column families in Cassandra or collections in MongoDB. Each table consists of a set of rows (items as in DynamoDB or Documents as in MongoDB). Each row is identified by a unique key. In addition, the table schema is dynamic so it could have a variable number of columns or attributes. Similarly, the supported set of data operations is usually small and comes in the form of CRUD operations (Create, Read, Update, Delete).

Despite this simplicity, modern NoSQL data stores are still inflexible and lack many features. Examples include:

1. History of data operations and system logs is kept in flat files, an unfriendly format for system administrators who need to analyze them.

2. Data provenance, the derivation and ownership history of the user data, is not collected as part of the system design.

3. System metrics are not provided in flexible granularity.

All these features represent a form of metadata that can be collected during system operations. Throughout this work, we define metadata as the information associated with data that is not directly used for data operations (i.e. CRUD) or administration operations (e.g. Create or Alter).

We propose Wasef[1], a metadata system for NoSQL data stores. The system functions as a component of the data store and employs the underlying distributed infrastructure and storage capabilities to deliver its services. We

[1] Arabic word for ”Descriptor”

argue that Wasef should be an intrinsic part of the NoSQL store design for two reasons. First, it provides the infrastructure for building valuable features and improving internal operations. Second, it imposes a low performance overhead on the hosting system.

This work addresses two key challenges. First, it provides a general design for a metadata system that is amenable to integration with all NoSQL data stores regardless of their implementation differences. Second, it collects metadata without imposing heavy overhead on the underlying system. NoSQL data stores handle huge loads of data at a very high throughput. Therefore, they are susceptible to adding extra operations that might delay the critical read or write paths.

1.1 Technical Contribution

We make the following technical contributions in this work:

1. We provide design principles, storage layout, and APIs for a general metadata system called Wasef that works as part of the current NoSQL data stores.

2. We demonstrate our design by incorporating Wasef into Cassandra 1.2. The customized version is used to evaluate the performance and scalability of our system.

3. We provided three use-case scenarios for Wasef to show its advantages: (a) A flexible implementation of the column drop operation that also solves a major issue in Cassandra 1.2. (b) A fully functional data provenance feature. (c) A verification tool for node decommissioning from Cassandra cluster.

4. We provide a comprehensive evaluation of the performance and scalability of Wasef and the use-case scenarios built on top of it.

The rest of the thesis is organized as follows. We describe the related work and position our system offerings within it in chapter 2. Then, we layout the background about the systems and the concepts covered through the thesis (chapter 3). Chapter 4 presents the system design. In chapter 5, we discuss the implementation challenges. Chapter 6 describes the design and implementation of the use-case scenarios. The system is evaluated in chapter 7. We conclude by chapter 8.

CHAPTER 2

RELATED WORK

Since the term metadata was coined by Philip Bagley to describe containers of data in the context of programming languages [6], it evolved to include multiple definitions. Bretherton and Singley [7], identified two types of metadata. First, structural metadata which describes database entities (e.g. tables) and the hierarchical relationships between them. Second, descriptive metadata which refers to descriptive data about the data items stored in the database. The National Information Standards Organization (NISO) defines a new type of metadata called administrative metadata [8]. It covers the data that can be employed to manage the system resources. Our system, according to these definitions, collects descriptive and administrative metadata for NoSQL data stores.

Metadata systems can be internal to the database system [9, 10] or external [11–14]. External metadata systems are used extensively for managing business metadata repositories [14] and maintaining scientific metadata in grid computational environment as in [15–18]. Our system works as a component of the NoSQL data stores.

Structural metadata is maintained internally in relational databases (i.e. the catalog) and NoSQL databases [1] to manage the database entities. Our system collects different type of metadata that describes client data and the operations performed on it.

Several works argued for providing metadata as a feature of the cloud data stores including [19–23]. Muniswamy-Reddy et. al. [19] suggested three protocols for a metadata system augmented to the cloud service clients to collect and maintain metadata about the stored objects. Our system is different since it works as part of the main data store. Therefore, it causes less overhead when moving the metadata, and it ensures consistency between the data and the metadata. These issues are hard to solve when the metadata system is part of the client.

Trio [10] is a data management system design to work on top of relational database (e.g. Postgres). The system supports new features such as data uncertainty and specialized queries for data provenance. However, this system does not support scalability which is one of our system's offerings.

Data provenance, one kind of metadata, received considerable attention recently. This include provenance collection and management in scientific workflows [24–27], monitoring system operations [22, 28–30], and database queries [31–35]. Buneman and Tan [36] studied query provenance for relational databases and divided it into two types. First, Where Provenance, that describes the source records of the query results. Second, Why Provenance, which justifies the query results by describing the performed operations and the relationship between source records. Our system takes advantage of the simple data model of the NoSQL data stores to provide both types of provenance.

Research in Provenance for key-value stores is relatively new. Examples include [9, 19, 37]. Kulkarni [37] suggests a provenance model for Cassandra data store, called KVPMC, similar to our model. KVPMC collects provenance information on request, provide client access, and can store provenance data internally or externally. However, there are many differences between the two systems. First, our system suggests a generic model for modern NoSQL data stores not only for Cassandra. Second, our system collects administrative and operational metadata in addition to the data provenance. Third, KVPMC collects provenance at the client side when the operation is done while our system integrates with the NoSQL datastore and collect provenance during the operation. Finally, our system imposes less overhead and provides linear scalability while KVPMC does not.

Many commercial and open source NoSQL datastores support a form of client-accessible metadata including [38–40]. For instance, Amazon S3 [39] provides two forms of metadata: system-defined metadata (e.g. object size and creation time) and user-defined metadata which can be assigned to S3 objects at the insertion time. These metadata services are inflexible. For instance, the size of the metadata is restricted to tens of kilobytes, and some basic features like searching the existing metadata are not supported. Our system is flexible since it treats the metadata as any of the client data. Therefore, there are no limitations on the size of the metadata and all data operations are also available for it.

CHAPTER 3

BACKGROUND

3.1 Apache Cassandra

Cassandra [3] is a distributed open source NoSQL data store. It has a peer-to-peer design that provides continuous availability, tunable consistency, and replication across the cluster nodes. It is designed to handle massive data loads where it scales out linearly to a large number of nodes across multiple data centres [41]. Cassandra features a flexible data model and a SQL like query language [42] with a fast response time [43]. This section summarises the original Cassandra paper [3] and the Datastax online documentation [44].

3.1.1 Data Model

In Cassandra, all data operations including partitioning, reading, writing, and compaction are performed at the row-level. Rows are grouped into column families or tables such that the rows within the same column family are identified by primary keys of the same type. A keyspace or schema, is a logical grouping of column families. For instance, system tables are grouped under a single keyspace called `System`.

Row primary key is divided into two parts. The first part is called the partitioning key since it is used to decide the hosting replica of the row. The second part is called the clustering key, which is used by the internal storage service to cluster the non-key columns as a single entity associated with the partitioning key.

3.1.2 Cluster Operations

Cluster membership in Cassandra is managed by a peer-to-peer gossip protocol [3] that allows the nodes to periodically exchange their locations. The protocol is also used for failure detection and dissemination of other state control information.

The data distribution across the cluster is determined by the Cassandra partitioner based on the concept of consistent hashing [45]. In consistent hashing, the output range of the partitioner (e.g. MD5 is used in `RandomPartitioner`) is treated as a ring. Cassandra divides the ring into small ranges called virtual nodes and assign them randomly to the cluster nodes as illustrated in figure 3.1. The hosting node for a data entry (i.e. row) is determined by finding the position of the hash value of the key in the ring.

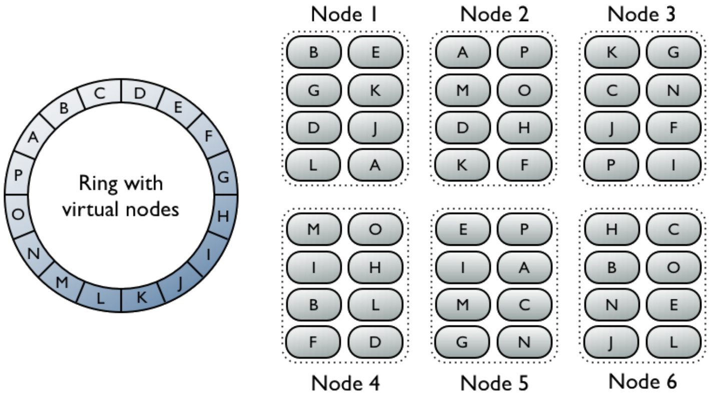


Figure 3.1: The right part shows a Cassandra hashing ring marked with virtual nodes. The left part shows the virtual nodes distributed randomly across the physical nodes of the cluster. Taken from [44].

The hosts of the data replicas are decided by the replication strategy. For instance, `SimpleStrategy` places the first replica according to the position determined by the partitioner, while the rest of the replicas are positioned by moving clockwise on the hashing ring to the next nodes. Cassandra also provides `NetworkTopologyStrategy` that efficiently places nodes on multi-data center clusters. The number of replicas is configurable and determined at the keyspace level.

3.1.3 Client Requests

Client Requests are divided into read and write requests. The node that receives the client request is called the coordinator, which can be any node in the cluster since all the nodes are peers. The coordinator relies on the partitioner and the replication strategy to determine the replicas that host the record.

Write requests are handled as follows. First, the coordinator forwards the request to all the replicas. Second, the replicas acknowledge the local writing of the entry to the coordinator after finishing the operation. Third, the coordinator acknowledge the results back to the client after receiving a subset of the replicas acknowledgements determined by the consistency level. Figure 3.2 shows a consistency level of one. Which means the coordinator will reply back to the client after receiving the first write acknowledgement.

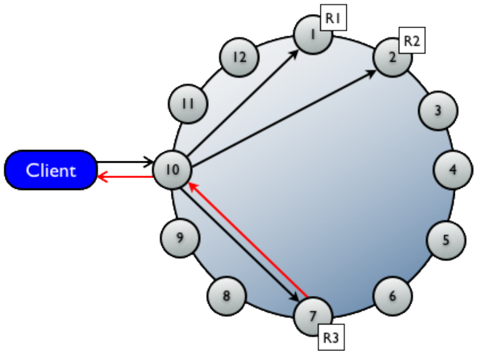


Figure 3.2: Client write request to Cassandra cluster. For a consistency level of one, the coordinator forwards the request to all replicas and wait only for the first reply. Taken from [44].

Read requests are forwarded to a subset of the replicas that is decided according to the consistency level. If the consistency level is higher than one (e.g. Quorum), multiple values will be received at the coordinator form the replicas. The coordinator replies to the client with the latest version of the data.

3.1.4 Internal Operations

Write operation is performed in several internal steps. Initially, when a write operation is received, it is written immediately to a commit log (for data durability against node failures) and to a memory structure called memTable that serves a single column family. The size of the memTable keeps increasing with every write operation until it reaches a configurable limit. After that, the memTables are flushed into SSTables on the disk using sequential I/O. Note that SSTables are immutable once they are written to the disk for fast operations, which implies that column families can be represented on the disk by multiple SSTables. Finally, Cassandra performs disk compaction periodically to reduce the size of the accumulating SSTables. Compaction merges the conflicting versions of SSTable entries by taking the record with the latest timestamp.

Internal **read operations** starts by checking the bloom filter associated with the SSTable to find if there is data for the requested key. The bloom filter is a probabilistic data structure that tests if an element is a member of a set, and it only allows false positives results [46]. If the key is found in the bloom filter, Cassandra checks the key cache to get the index of the key in the SSTable. In case of cache miss, a sample of the index, called IndexSummary, is checked to determine the approximate location of the key. When the index is finally obtained, the value is retrieved from the SSTable and returned back to the coordinator. The read workflow of a compressed sstable is illustrated in figure 3.3.

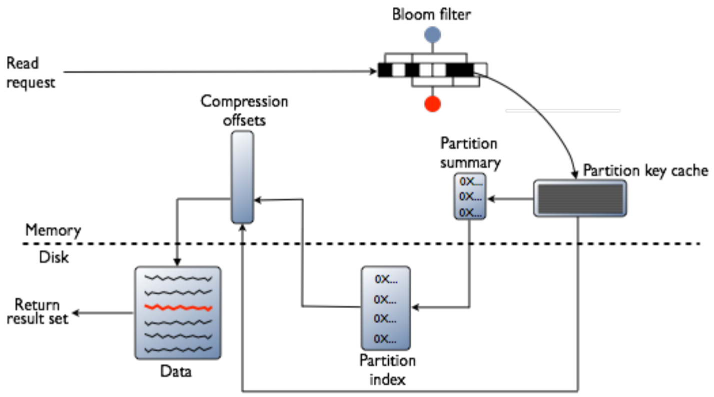


Figure 3.3: Steps of an internal read operation from a compressed SSTable. Taken from [44].

3.2 Data Provenance

Data provenance, or lineage, is the history of an item, which includes its source, derivation, and ownership. Provenance increases the value of the item since it proves its authenticity and reproducibility. One of the applications of data provenance is to track the workflow of scientific experiments, where the experimental setup and steps are recorded to help verifying and reproducing the final results.

Buneman and Tan [47] identified two types of provenance. First, the coarse-grain provenance, in which the complete history of the tracked data set or workflow is recorded. Second, the fine-grained provenance, which is the collection of partial derivation of the results. Fine-grained type becomes important when the workflow is complicated or partially unavailable.

In relational databases, query provenance includes the database records that participated in producing the results (the source), the relation between records, and the performed operations. The sophistication of SQL queries represents the key challenge for this area of data provenance.

The data model of the NoSQL data stores is relationless which simplifies identifying provenance data and collecting it. However, the massive size of the data handled by the NoSQL data store complicates the management and the accessibility of the provenance data. The problem can be alleviated by collecting fine-grained provenance and using the data store infrastructure for providing client access. We adopt this approach in this work.

CHAPTER 4

SYSTEM DESIGN

In this chapter, we present the architecture of Wasef. Our design addresses the key challenges presented in the introduction as follows. First, it exploits the fact that most NoSQL data stores share a simple data model to provide a generic storage architecture that fits most of data stores. Second, it integrates with the existing functionality of the underlying data store to provide light weight metadata operations.

We start by laying out the principles used to guide our design (section 4.1), which emphasize lightweight operations and minimal collection of metadata. Then, we define the system components in section 4.2. Section 4.3 describes the workflow of Wasef starting by the registration of the metadata targets and ending with external APIs to exploit the collected metadata. Finally, section 4.4 lists system's internal and external APIs. Figure 4.1 illustrates the system architecture.

4.1 Design Principles

The following principles guided the design of Wasef:

Modularity and integration with the existing functionality:

NoSQL data stores are superior to their counterparts in providing distributed, highly available, and fast storage capabilities. Being a module within the NoSQL data store, the metadata system should integrate with the underlying infrastructure without affecting the existing functionality. This ensures that the system has minimal effect on the data store performance.

Flexible granularity of the collected metadata:

To achieve a generic architecture that fits current NoSQL data stores,

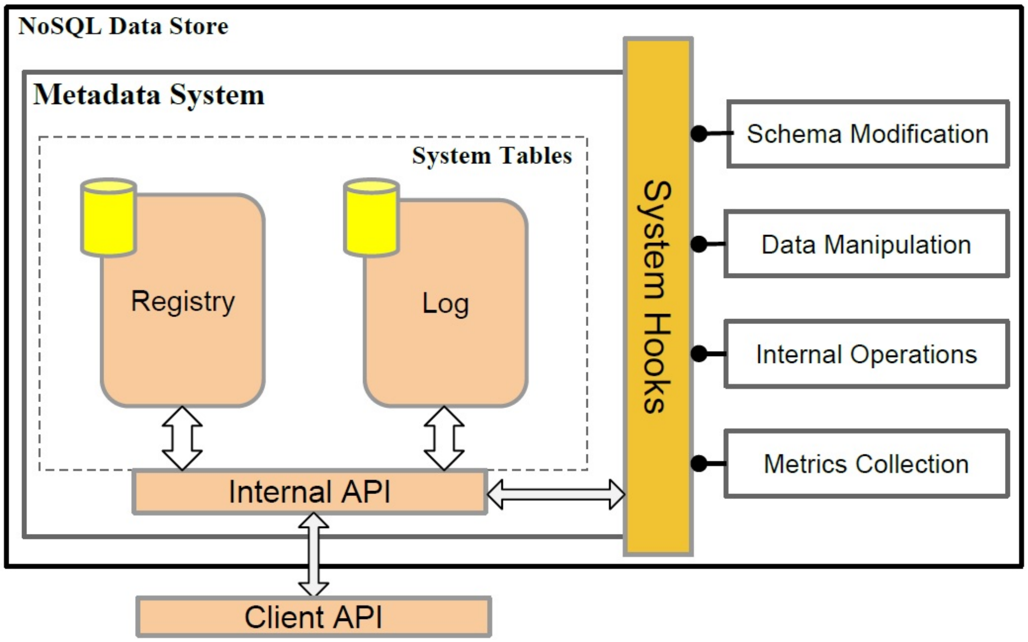


Figure 4.1: The architecture of Wasef

the design should be flexible to collect and store metadata about objects and operations of different types and granularities. Those include the time and description of the performed operations, object names, and ownership information.

Minimal collection of the metadata:

Due to the enormous size of the data and operations handled by NoSQL data stores, the continuous collection of metadata about every operation imposes a huge overhead on system. Collecting metadata for a selected set of operations will remedy this problem.

Accessibility of metadata by internal and external clients:

The selectively collected metadata proves useful, as we will demonstrate in chapter 6, for internal server operations (e.g. column drop) and external clients (e.g. data provenance). Therefore the system should provide them with suitable APIs for accessibility.

4.2 Architectural Overview

As illustrated in figure 4.1, Wasef consists of the following components:

Registry: A system table used to register metadata target names and operations in the system. Metadata will be only collected about registered objects and operations.

Log: a system table where the collected metadata is stored.

Internal API: A software component that integrates with the data store. It provides two functionalities. First, it implements the main logic of Wasef. Second, it exposes a set of interfaces for internal data store components to exploit Wasef.

System hooks: data store-dependent implementations that call the internal API to report the registered metadata to Log table.

External API: a set of functions exposed to the data store users to allow them to register objects and operations in the system, and exploit the collected metadata.

4.3 Operational Workflow

The collection of metadata starts by registering metadata targets and operations in the Registry table. Targets are names of data entities, such as tables or rows, or internal components of the data store, such as SSTables or cluster nodes. Operations are names of events occur to targets that will trigger the metadata collection such as schema modification, row insertion, or node decommissioning. The registration of the targets and operations is done via an internal or external API.

The actual collection of the metadata is done at different parts of the data store by the System Hooks. For instance, an `Alter Table` statement can be instrumented to report the name of the table, the old and new values, and other status information such as the user who modified the table. Several examples are provided in Chapter 5 to demonstrate how different parts of the system can be instrumented to exploit Wasef.

Finally, the collected metadata accumulates in Log table. The internal and external clients of the metadata can query that table via the exposed APIs.

4.4 Metadata API

In this section we details the internal and external API exposed by Wasef.

4.4.1 Internal API

The internal APIs provide basic data manipulation operations for Registry and Log tables. The APIs for accessing the Registry table are:

```
Registry.insert(target, dataTag, adminTag)
Registry.delete(target, dataTag)
Registry.query(target, dataTag)
```

Where the `target` and `dataTag` parameters are mandatory in all the functions. The following APIs are provided for accessing the Log table:

```
Log.add(target, time, client, dataTag, adminTag, value)
Log.drop(target, startTime, endTime, client, dataTag)
Log.query(target, startTime, endTime, client, dataTag)
```

Before `Log.add` stores metadata records in Log table, it validates `target` and `dataTag` parameters against the metadata registry. In addition, when the `adminTag` parameter is provided, the related metadata is collected before completing the operation. All the parameters of `Log.query` function are optional, except `target`, to provide flexibility for querying the metadata log.

4.4.2 External API

It is a simplified version of the internal API. The following functions are exposed to the external clients:

```
register(target, dataTag, adminTag)
unRegister(target, dataTag)
queryAll(target, dataTag, client)
queryLatest(target, dataTag, client)
```

Although query operations are limited to retrieving either all or latest metadata for a specific target, they can be easily extended through the data store data operations.

CHAPTER 5

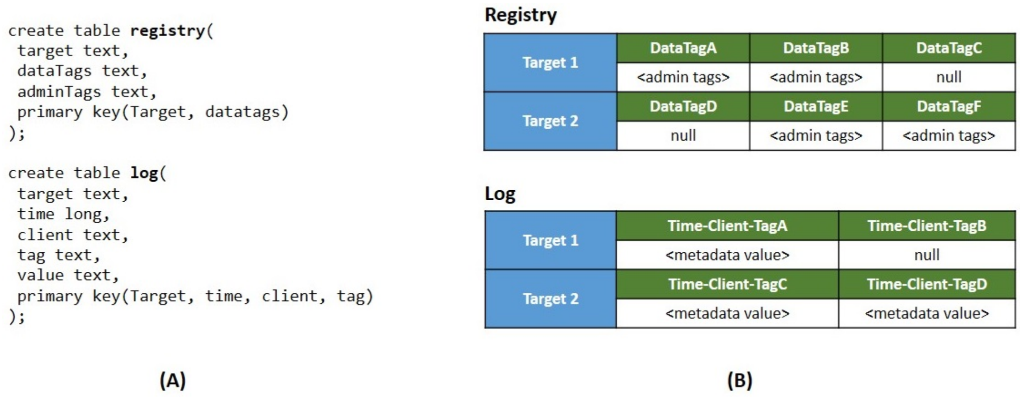
IMPLEMENTATION

We provide a customized version of Cassandra 1.2 augmented with Wasef. Our modifications come in two forms: First, the core system components such as storage operations, metadata validation, and the internal APIs are implemented as a separate package. Second, the data store-dependent hooks are injected in the source code where necessary. In this chapter, we describe the design layout (section 5.1), and the metadata targets and operations supported by our system (section 5.2). We conclude the chapter by a discussion of the implementation challenges (section 5.3).

5.1 Storage Layout

Following Cassandra's convention of storing system related data, we organize the metadata tables under `system_metadata` system keyspace. Choosing this layout has many advantages. First, it achieves our first design principle (section 4.1) of exploiting the existing functionality of the underlying system such as replication, caching, and fast access. Second, using a system keyspace provides a read-only protection for the metadata schema, and makes it directly available after system bootstrapping.

Figure 5.1.A illustrates the schema definitions of the metadata tables. The `target` field corresponds to a data store entity name and `dataTag` specifies an operation over the target. Both `target` and `dataTag` are required to register a metadata entry. For instance, a table row identified by its primary keys can have `ROW_INSERT` as its `dataTag`. Section 5.2 provides a suggested list of targets and operations for Cassandra data store. The optional `adminTag` field is used to specify the name of operations and data that are not directly related to the target. For instance, various metrics about the system such as read latency, current compaction stage, and others can be specified. Finally, the `client` field in the Log table reports the ownership information of the metadata target, which is the user that is authorized to do the operation in the case of Cassandra data store.



```
create table registry(
 target text,
 dataTags text,
 adminTags text,
 primary key(Target, datatags)
);

create table log(
 target text,
 time long,
 client text,
 tag text,
 value text,
 primary key(Target, time, client, tag)
);
```

(A)

Registry

Target 1	DataTagA	DataTagB	DataTagC
	<admin tags>	<admin tags>	null
Target 2	DataTagD	DataTagE	DataTagF
	null	<admin tags>	<admin tags>

Log

Target 1	Time-Client-TagA	Time-Client-TagB
	<metadata value>	null
Target 2	Time-Client-TagC	Time-Client-TagD
	<metadata value>	<metadata value>

(B)

Figure 5.1: Wasef storage schema: (A) Shows the CQL queries used to create the metadata tables. (B) Illustrates an example for the internal storage layout of the metadata tables. Note how column names are composed of the clustering primary keys names.

The primary keys of the metadata tables are carefully chosen and ordered to achieve two goals:

1. Optimizing the storage layout for low read latency: The `target` key works as the partitioning key for both tables (refer to section 3.1 for background information) while the rest of the keys serve as column identifiers. Grouping the metadata related to one target within the same row orders the fields lexicographically and ensures they reside in the same Cassandra node which leads to faster reading. Figure 5.1.B illustrates an example.

2. Flexible querying of `Log` table: In Cassandra, the where clause of the select statement can only be used to filter the primary keys of the table. Including more fields in the primary key increases the querying flexibility.

5.2 Supported Metadata Targets and Operations

We used the CQL 1.2 documentation [42] to compile a list of metadata targets and operations for Cassandra as illustrated in table 5.1. The create operation is omitted from the list since it requires registering non-existing objects to the registry. It is also important to note that adding extra targets to the list is possible but requires adding system-dependent hooks to collect the required metadata.

target	Identifier	Operations	Metadata
Schema	Name	`Alter` `Drop`	Old and new names, replication map

Table	Name	`Alter`
		`Drop`
		`Truncate`
			column family name, column names and types, compaction strategy, ..

Row	Partitioning keys	`Insert`
		`Update`
		`Delete`
			key names, affected columns, TTL and timestamp

Column	Clustering keys and column name	`Insert`
		`Update`
		`Delete`
			key names, affected columns, TTL and timestamp

Node	Node ID	On request	Token ranges

Table 5.1: Supported metadata targets and operations

5.2.1 Naming convention

To register and lookup metadata the following convention is used:

```
<Keyspace name>.<Column family name>.<Comma separated
list of partitioning keys>.<Dot separated list of cl-
ustering keys>.<Non-key column name>
```

Please refer to the background (section 3.1.1) for an explanation of partitioning and clustering keys that identifies rows and columns in Cassandra.

5.2.2 Cassandra Metrics

The optional `adminTag` field in Registry table allows collecting any of the metrics offered by Cassandra [48] along with the metadata. Associating system statistics and metadata provides system administrators with detailed system health statistics. For instance, collecting cache hit rate along with row insertion metadata measures how trending is that row which can be used to tune the data store schema. System metrics are collected during `log.add` operation.

5.2.3 Data Ownership

Once the authentication feature in Cassandra is enabled, metadata `log.add` operation starts to report the session owner of the performed operation. In order to implement this feature, we modified the writing path to propagate the ownership information down to the metadata logging. The ownership information is very important for data provenance feature.

5.3 Implementation Challenges

Reporting metadata to the storage begins with validating new entries against the metadata registry. In case of validation success, the system issues write operations to the metadata log. Since metadata targets are designed to be of a fine-grained granularity, collecting metadata about targets subject to high frequency operations (e.g. rows) raises a performance overhead challenge. To address this challenge, we employ the internal infrastructure of Cassandra as the following:

Fast registry lookup:

> Looking up registered metadata is done by issuing internal `fetchRows` operation. This is the way Cassandra internally fetches the rows for update and delete operations. Cassandra reduces read latency by sending read requests to the closest nodes based on their proximity (the network distance to the source node). We use `ANY` consistency level that limits the number of requested copies to one. In addition, row caching is enabled at the source nodes to reduce the lookup time. We evaluate the read latency of metadata validation in section 7.2.

Lightweight log insertion:

Cassandra uses the SEDA design [49], which divides the work into stages with a separate thread pool per stage. This design is exploited to improve the metadata write efficiency by injecting the writes into the Mutation stage in a separate thread. The write consistency level is set to `ANY` to avoid long waits. The overhead imposed by the metadata writes is also evaluated in section 7.2.

CHAPTER 6

LEVERAGING THE METADATA SYSTEM

The metadata system APIs open the door for internal and external clients to leverage the power of the collected metadata to address system pain points and provide new features. To demonstrate this, we implement three use-case scenarios selected from three different domains. First, we provide a flexible column drop operation that solves the issue in the standard Cassandra 1.2 (section 6.1). Then, we describe the design and implementation of our verification tool for node decommissioning in section 6.2. Finally, we describe the data provenance feature in section 6.3.

6.1 Allowing Flexible Column Drops

In the first use-case we solve one of Cassandra's major bugs using the metadata system. It shows an example of how the system can be leveraged to improve data operations.

The JIRA bug number 3919 [50] reports a problem in the column drop operation. According to the bug, dropping a column removes its definition from the table schema but keeps the column data stored in the system. Therefore, if another column is added to the table with a name equal to the dropped one, the data of the old column will be shown when the new column is selected.

We present a generalized and flexible column drop implementation that leverages the metadata collected by Wasef. Figure 6.1 illustrates the state diagram of our implementation. In this implementation, dropping the column for the first time is considered as a tentative drop that removes the schema definition of the column but keeps the data. Adding the column after that retrieves the original status of the system. However, if the tentative drop is confirmed by a second column drop operation the column data is permanently purged. We provide a grace period for re-adding tentatively deleted column. When the grace period expires, the column data is also purged permanently.

When a column is dropped using `Alter table` operation for the first time, a metadata called `AlterColumnFamily_Drop` is inserted in Registry table to start monitoring the second drop operation. In case the column is re-added the metadata is deleted. However, if the column is dropped again, a metadata log entry that contains the column name and the time of the drop operation is inserted in the Log table. The `AlterColumnFamily_Drop` metadata and the log entry are used together to identify the permanently dropped columns during select operations, so they are filtered from the final results.

Cassandra provided a fix for the column-drop issue in a later release (Cassandra 2.0) that also relies on keeping the history of the dropped columns. However, the proposed fix keeps that history in a specialized hash map attached to the column schema and maintain a copy of that hash map at each node in the cluster. Our solution uses a generic infrastructure that can be applied for similar type of problems, and it maintains the history in a subset of the cluster nodes based on the configurable replication factor of the metadata schema.

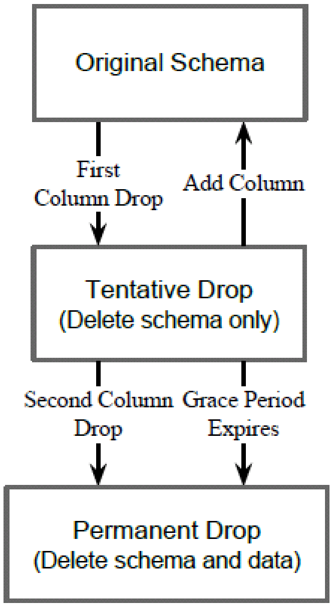


Figure 6.1: Column Drop Operation

6.2 Node Decommissioning

In this section we implement a verification tool for the node decommissioning operation which relies on the metadata collected by our system. The tool verifies for system administrators that all the data existed in the decommissioned node are safely moved to the other nodes in the cluster. Without this feature, the administrator needs to manually count the token ranges assigned to the decommissioned node, and then check if every one of them is reassigned to one of the other nodes in the cluster.

Node decommissioning is one of the operations provided by Cassandra's NodeTool utility. When it is performed from one of the cluster nodes, the node streams all of its data to the other nodes in the cluster and unbootstrap. The new destinations of the streamed data are calculated as follows. First, the token ranges related to each non-system table are collected (refer to section 3.1.2). Second, the partitioner and the replication strategy are used to decide the new replica for each token range. Third, all the collected information is passed to the file streamer to move the SSTables to the intended destinations. Finally, the node is retired from the cluster.

This operation becomes critical when there is only a single copy of the data held by the decommissioned node (i.e. the node holds part of a keyspaces with replication factor of one). In this case, verifying that all the data has been safely moved is important. Unfortunately, Cassandra does not provide a verification tool.

We exploit the metadata collected by Wasef to verify node decommissioning as follows. During the decommission command, we use `Log.add` API (section 4.4) to store the new replica for each token range hosted by the decomissioned node in the metadata Log with a metadata called `decommission`. Then, when the decommissioned node leaves the cluster, the operation can be verified using the command:

```
nodetool decommission -verify <decommission node IP>
```

This command retrieves the metadata records of the decommissioned nodes (using `Log.query` API) and verifies that all the ranges are currently available in the system using the partitioner and the replication strategy of each keyspace. Section 7.6 evaluates the overhead of this operation.

6.3 Providing Data Provenance

Modern NoSQL data stores do not support data provenance by design. In this section, we show that our metadata system fills this gap for the underlying data store.

Collecting query provenance is provided by design in our metadata system. As we show in section 5.2, we collect the following provenance data about each operation in Cassandra (refer to table 5.1 for a comprehensive list of operations):

1. The full name of the target of the data operation (section 5.2.1). For example, dropping a table called `User` located in `Test` keyspace results in logging `Test.User` as the full name.

2. Operation name. For example, `Alter_Add_ColumnFamily` is used to indicate adding a new column to a column family.

3. Operation time (i.e. the timestamp of the operation).

4. The authenticated session owner name.

5. The results of the operation. This varies depending on the operation. For instance, when a new column is added the name of the column and its attributes are logged, and when a column name is modified the old and the new names are logged.

This information covers two types of query provenance (related work, chapter 2): First, the Where Provenance, which is the records from which the query results are derived. Second, the When Provenance, which is the justification of the results by reporting the operations that produced them.

Provenance data are treated like any of the system data. This implies that the replication, scalability, and accessibility features of the NoSQL data store are also available for the provenance data. In addition, we provide external APIs (section 4.4) to provide easy access for the clients.

Our system does not provide automatic garbage collection for old provenance data. However, system administrator can use the delete APIs provided by our system to manually delete old provenance data entries based on their timestamps.

CHAPTER 7

EVALUATION

In this chapter, we evaluate various aspects of Wasef and the use-case scenarios provided in chapter 6. The YCSB [43] benchmark is utilized in many of the chapter tests to measure the latencies introduced by the metadata system (section 7.2), the scalability (section 7.3), and the overhead of collecting provenance data (section 7.5). In addition, we design other tests to evaluate column-drop operation (section 7.4) and node decommissioning verification tool (section 7.6).

We answer two main experimental questions throughout the chapter:

1. What is the performance cost of adding the metadata system and the features built on top of it on Cassandra? This includes read and write latencies, and the overall throughput.

2. How does Cassandra scale with the size of the collected metadata?

7.1 Experimental Setup

In our experiments, YCSB tests use six Amazon EC2 m1.large machines, each one has 2 virtual CPUs (4 ECUs), 7.5 GB of RAM, and 480GB of ephemeral disk storage. We run the YCSB client from a separate machine that has the same configuration setup as the rest of the machines. The machines use Ubuntu 12.04 64-bit operating system with swapping off as recommended for Cassandra production installation. In addition, we use zipfian distribution to load and run all the YCSB tests. The data set size varies based on the experimental need.

7.2 Metadata System Throughput

This experiment evaluates the overhead introduced to Cassandra after adding the metadata system. We compare a standard Cassandra version 1.2.9 with a customized version augmented with the metadata system. The evaluation focuses on the update and read latencies as practical measures of Cassandra performance.

A series of 15 YCSB runs are conducted for each of the Cassandra versions using a heavy-update workload (50% update and %50 read). The update and read latencies are measured as the throughput is increased while fixing the number of nodes and the data set size. The technique for increasing throughput is adopted from the YCSB benchmark, where the number of YCSB working threads is continuously increased (up to 300 threads) to generate more throughput until the system is saturated.

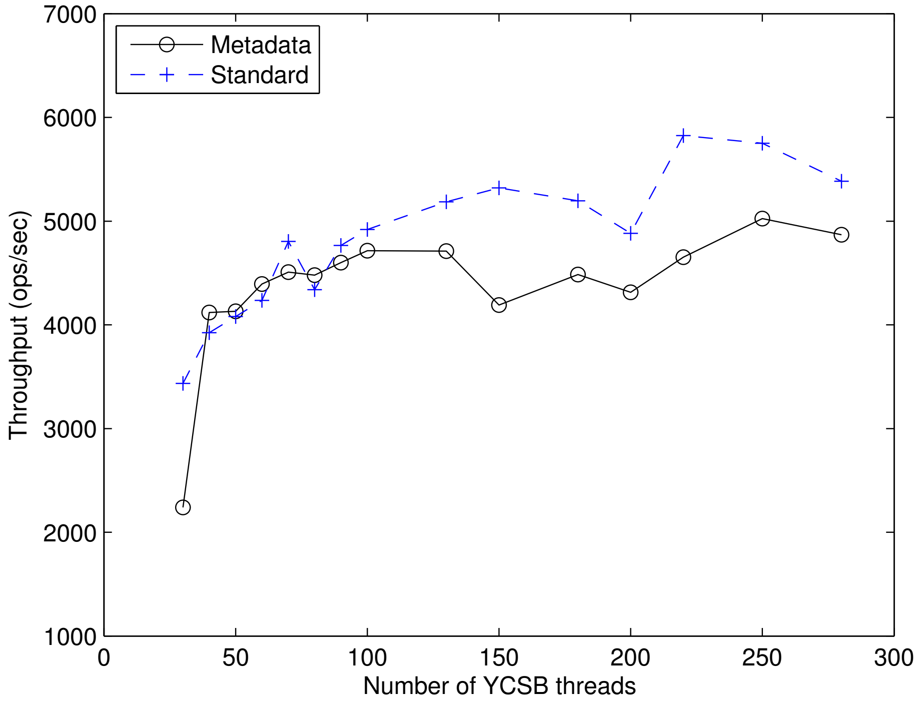


Figure 7.1: Throughput against number of clients: Throughput comparison between the standard Cassandra 1.2.9 and a customized version augmented with the metadata system. The experiment uses a cluster of six machines and a data set size of 12 GB. Each point in the graph is the average ops/sec of one million YCSB client requests. The average difference between the two lines is 9%.

Figure 7.1 illustrates the throughput results measured in operations per second. Our system shows a comparable throughput to the standard version when the number of client threads is moderate. However, when the system is overloaded by a large number of threads it shows a slight degradation of 9%. The degradation is related to the extra operations performed by the system to collect and record the metadata.

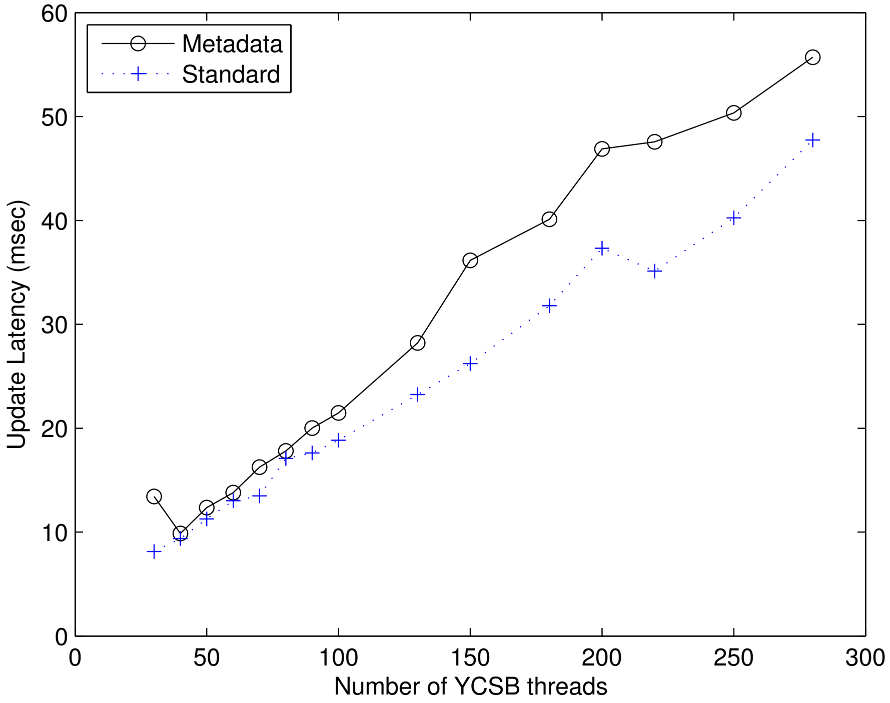


Figure 7.2: Update latency against number of clients: Update latency comparison between the standard Cassandra 1.2.9 and a customized version augmented with the metadata system. The experiment uses a cluster of six machines and a data set size of 12 GB. Each point in the graph is the average ops/sec of 500 thousand YCSB client requests.

Figures 7.2 and 7.3 compares the update and read latencies of the metadata version of Cassandra with the standard version. As in the throughput results, the update latency figure shows that the metadata system is comparable to the standard version when the number of threads is moderate. However, the latency slightly degrades under heavy load. On the contrary, the read latency of the metadata system is very close to the standard version with only 3% degradation on average. This is because our system does not collect metadata throughout the critical path of the read operation.

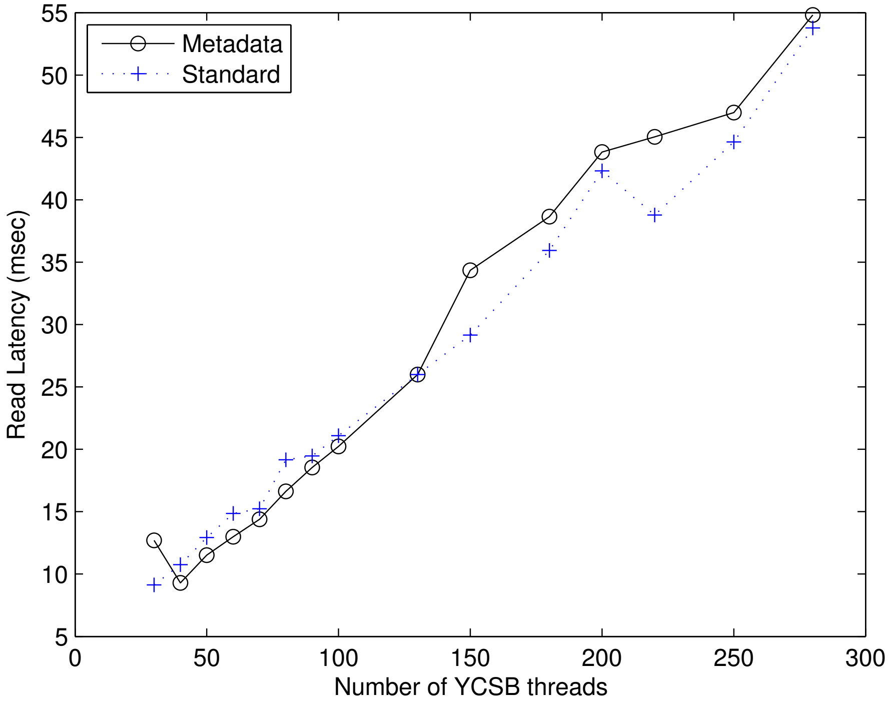


Figure 7.3: Read latency against number of clients: Read latency comparison between the standard Cassandra 1.2.9 and a customized version augmented with the metadata system. The experiment uses a cluster of six machines and a data set size of 12 GB. Each point in the graph is the average ops/sec of 500 thousand YCSB client requests.

7.3 Scalability

This test evaluates the impact of metadata system on Cassandra scaling capabilities. We use the scalability test offered by the YSCB benchmark which linearly increases the number of nodes in the cluster while keeping the size of the data set and the system load proportionally constant. For each run, the size of the cluster is increased by two nodes, the data set by four GB, and the load by 50 threads.

Figure 7.4 compares the update latency of the standard cassandra version against the metadata version as we scale the system out. The results clearly shows that Cassandra maintain linear scalability after adding the metadata system.

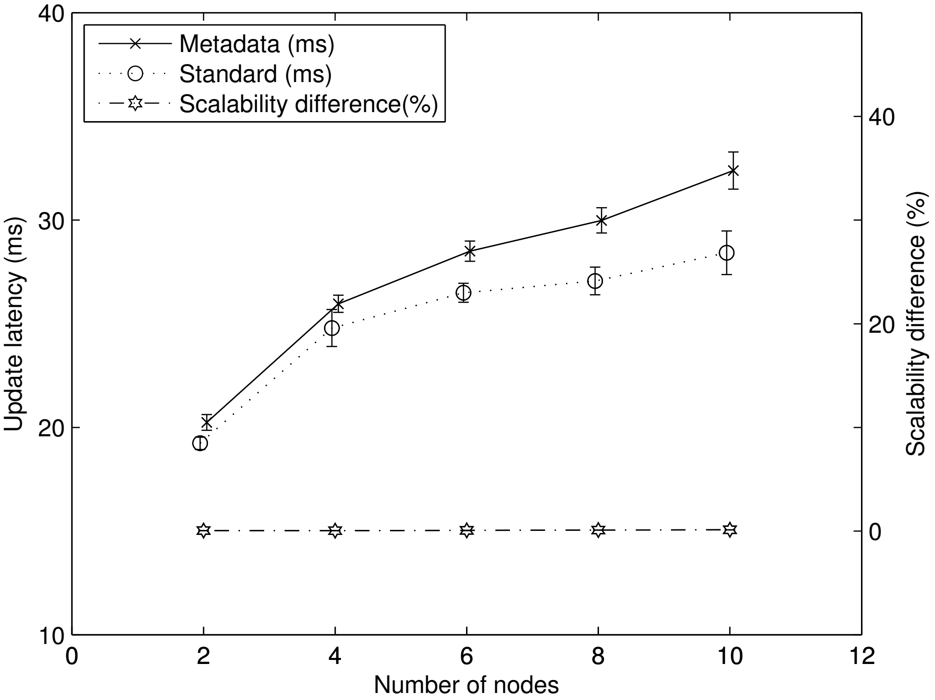


Figure 7.4: Scalability against cluster size: A scalability comparison between the standard Cassandra 1.2.9 and a customized version augmented with the metadata system. The experiment increases the number of cluster nodes linearly while keeping the data set size and YCSB load relatively constant. Note that the update latency of the standard version is shifted by 0.05 to the left, and for the metadata version by 0.05 to the right to show the standard deviation bars clearly. Also note that the scalability difference is not a straight line (it varies within 0.1%)

7.4 Column-Drop Feature

In this experiment we evaluate the bug-fix we provided for the column drop feature (section 6.1). Since the YCSB benchmark does not offer schema modification tests, we designed a customized test that performs a set of 500 drop operation on and report the average latency. We compare our customized version of Cassandra with the standard Cassandra 1.2.9 using a data set size of eight GB. Note that Cassandra 1.2.9 implementation does not work properly (it only deletes the schema definition and keeps the data). Figure 7.5 illustrates the results.

The figure shows that the latency readings of our implementation are better than the standard latency in some cases. This is because the drop operation in our implementation marks the column for deletion in the metadata Log only while the standard version delete the schema definition of the column.

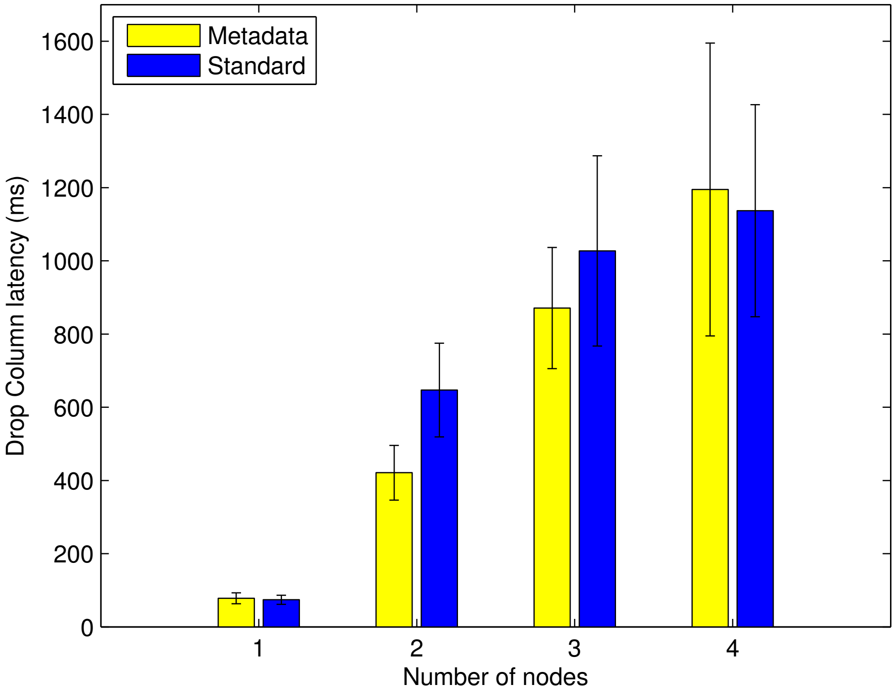


Figure 7.5: Column drop latency against cluster size: The latency of column drop operation for the standard Cassandra 1.2.9 compared to a customized version augmented with the metadata system. Each bar in the graph represents the average of 500 drop column operation performed by client running from a separate machine. The data set size is eight GB. The average latency overhead when our system lag the standard version is 5%.

7.5 Collecting Data Provenance

Database provenance information can be divided based on the type of the performed queries into two types: provenance about schema modification queries, and provenance about data manipulation queries. Section 7.4 provided an example performance measure for schema modifications queries. In this section we address data manipulation queries represented by update and read statements.

We already showed through the results of section 7.2 that validating metadata is the main source of the overhead in the system. Therefore, in this section we ask the following question: Are the read and update latencies affected by the size of the registered metadata?

To find the answer, the same experimental setup deployed in section 7.2 is used to measure the update and read latencies under various metadata sizes. Before running the YCSB client threads in each experiment, we register a randomly chosen subset of the database keys in the metadata system. This ensures that the operations performed on these keys are recorded in the metadata Log. For each metadata size, a series of 30 YCSB runs are conducted while increasing the load (the number of running threads) per run. The results are illustrated in figures 7.6 and 7.7.

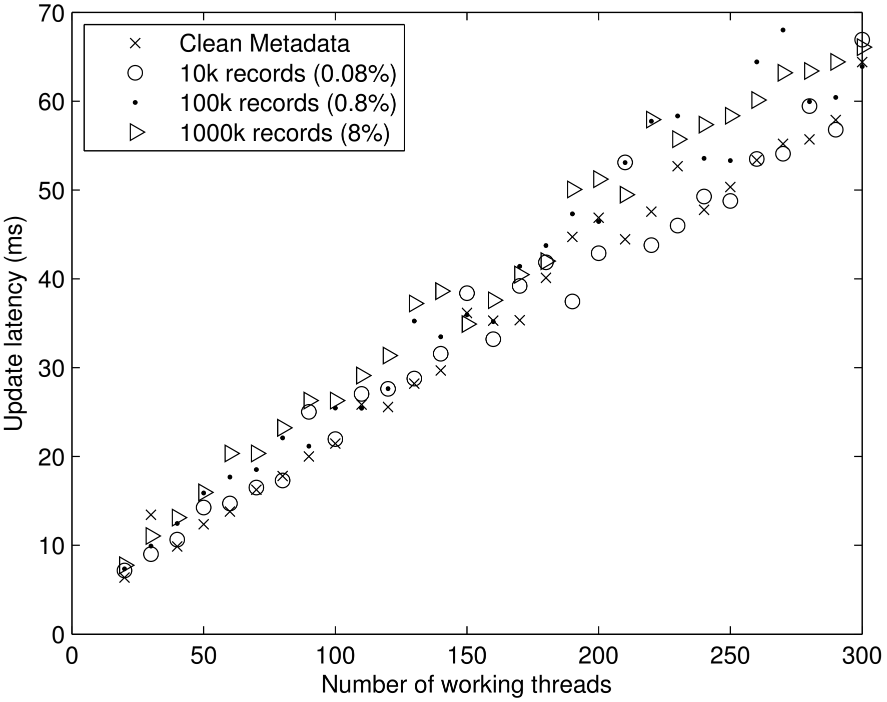


Figure 7.6: Update latency comparison between different metadata sizes registered in the metadata system. The experiment uses six machines with total data size of 12 GB.

The figures show clearly that the read and write latencies remain almost constant despite increasing the metadata size ten folds between executions. This is because the amount of work per each update (or read) remains the same regardless of whether a metadata is registered for the key or not.

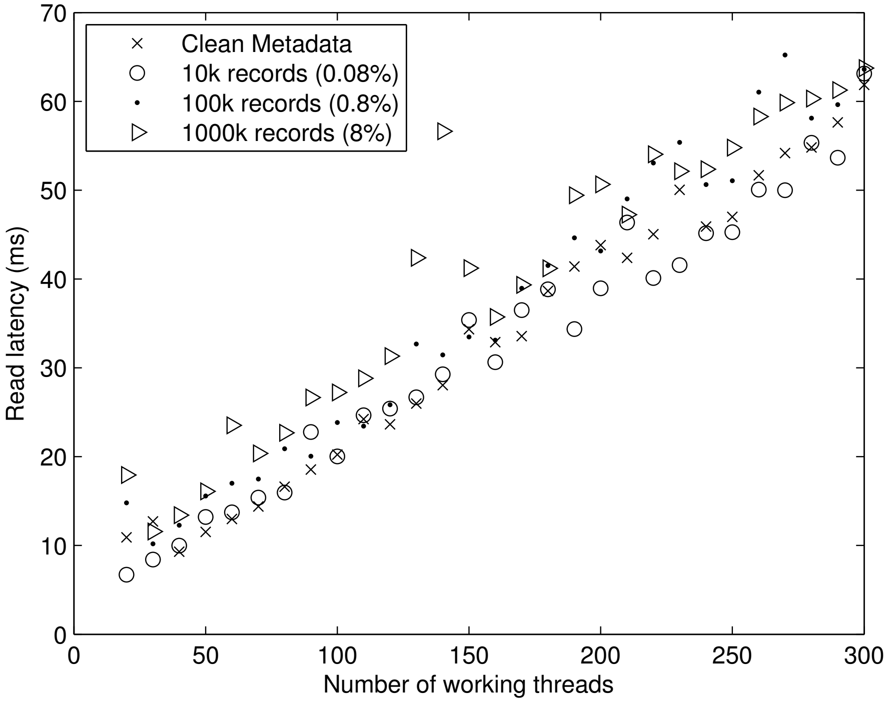


Figure 7.7: Read latency comparison between different metadata sizes registered in the metadata system. The experiment uses six machines with total data size of 12 GB.

7.6 Verifying Node Decommissioning

In this section we evaluate the overhead imposed by collecting metadata during node decommissioning operation. As describe in section 6.2, Node decommission consists of two steps. First, token ranges collection where the metadata is collected. Second, streaming the data to other nodes. We expect most of the running time to be spent on data streaming. Therefore, in order to measure the running time overhead during the first step we reduce the data set size from what we used in the previous experiments, and we vary the number of tokens per node.

We conduct the experiment using a cluster of four machines. The data set size is set to four GB. The number of token per machines is varied from 64 to 256. Figure 7.8 illustrates the results.

The results show two trends. First, the running time of the metadata version of Cassandra is very close to the standard version. We measure the average overhead as 1.5%. This difference is small and hardly noticeable by

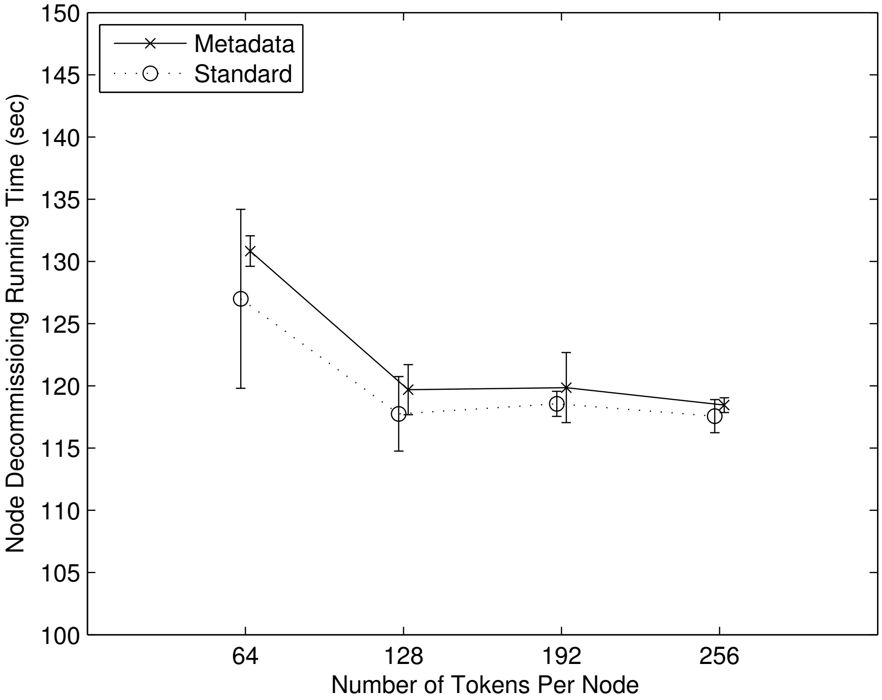


Figure 7.8: Running time for node decommissioning operation: The running time of the node decommission operation for standard Cassandra 1.2.9 compared to a customized version augmented with the metadata system. The data set size is 4GB. The average difference between the two lines is 1.5%. Note that the x-axis values are shifted by 0.03 to show the standard deviation bars clearly.

system administrators, specially if the data set size is very large.

The second trend is the inverse relation between the number of token and the decommission running time. Ideally, the running time should be constant given that the data set size is fixed. However, when the number of token ranges in the system is small, there is a greater chance for the data to be divided unequally between nodes. This causes a larger variation in the time required for streaming the data during decommissioning. When the number of token ranges becomes larger, the running time becomes smoother and gets close to the ideal case, which can be noticed in our results.

CHAPTER 8

CONCLUSION

We presented Wasef, a metadata system for the modern NoSQL data stores. The contribution of our work is twofold. First, we provided a generic architecture and API for the metadata that works for most of the current NoSQL data stores. Second, we implemented our design as a component of Cassandra data store without imposing heavy performance overhead and in a scalable manner.

In addition, we provided three use-case scenarios that leverage Wasef to provide new and flexible features. First, we implemented a flexible column drop operation that solves a major issue in the standard version of Cassandra. Second, we built a tool for system administrators to help them verify the correctness of node decommissioning operation. Third, we provided data provenance feature for client queries.

We evaluated the performance and scalability characteristics of our system and the features built on top of it. The evaluation shows that our system imposes low overhead on Cassandra througput of 9% (7.1) and read latency of 3% (7.3). We also showed that our system scales linearly with the number of cluster nodes and system load.

8.1 Future Work

Our work can be extended in the following ways:

1. Incorporating our system as a component of another NoSQL data store to improve its generic design.

2. Adding new metadata features such as ephemeral metadata, where metadata is collected for a limited period of time, and then automatically purges itself.

3. Providing further metadata use-case scenarios such as using our system to provide metadata service for scientific experimentation from a different field.

REFERENCES

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[3] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[4] "Project Voldemort." [Online]. Available: http://www.project-voldemort.com/voldemort/

[5] "MongoDB." [Online]. Available: https://www.mongodb.org/

[6] P. R. Bagley, "Extension of Programming Language Concepts," DTIC Document, Tech. Rep., 1968.

[7] F. P. Bretherton and P. T. Singley, "Metadata: A User's View," in *Seventh International Working Conference on Scientific and Statistical Database Management*. IEEE, 1994, pp. 166–174.

[8] N. Press, "Understanding Metadata," *National Information Standards*, vol. 20, 2004.

[9] K.-K. Muniswamy-Reddy and M. Seltzer, "Provenance as First Class Cloud Data," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 11–16, 2010.

[10] C. C. Aggarwal, "Trio: A System for Data Uncertainty and Lineage," in *Managing and Mining Uncertain Data*. Springer, 2009, pp. 1–35.

[11] E. Deelman, G. Singh, M. P. Atkinson, A. Chervenak, N. P. Chue Hong, C. Kesselman, S. Patil, L. Pearlman, and M.-H. Su, "Grid-Based Metadata Services," in *The 16th International Conference on Scientific and Statistical Database Management*. IEEE, 2004, pp. 393–402.

[12] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman, "A Metadata Catalog Service for Data Intensive Applications," in *ACM/IEEE Conference in Supercomputing*. IEEE, 2003, pp. 33–33.

[13] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, "Tango: Distributed Data Structures Over a Shared Log," in *The 24th ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 325–340.

[14] "Oracle Metadata Service (MDS) in Fusion Middleware 11g." [Online]. Available: http://www.oracle.com/technetwork/developer-tools/jdev/metadataservices-fmw-11gr1-130345.pdf

[15] M. B. Jones, C. Berkley, J. Bojilova, and M. Schildhauer, "Managing Scientific Metadata," *Internet Computing, IEEE*, vol. 5, no. 5, pp. 59–68, 2001.

[16] M. Xiong, H. Jin, and S. Wu, "2-Layered Metadata Service Model in Grid Environment," in *Distributed and Parallel Computing*. Springer, 2005, pp. 103–111.

[17] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good et al., "Pegasus: A Framework for Mapping Complex Scientific Workflows Onto Distributed Systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[18] P. Zhao, A. Chen, Y. Liu, L. Di, W. Yang, and P. Li, "Grid Metadata Catalog Service-Based OGC Web Registry Service," in *The 12th International Workshop on Geographic Information Systems*. ACM, 2004, pp. 22–30.

[19] K.-K. Muniswamy-Reddy, P. Macko, and M. I. Seltzer, "Provenance for the Cloud," in *FAST*, vol. 10, 2010, pp. 15–14.

[20] M. A. Sakka, B. Defude, and J. Tellez, "Document Provenance in the Cloud: Constraints and Challenges," in *Networked Services and Applications-Engineering, Control and Management*. Springer, 2010, pp. 107–117.

[21] O. Q. Zhang, M. Kirchberg, R. K. Ko, and B. S. Lee, "How to Track Your Data: The Case for Cloud Computing Provenance," in *The Third International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2011, pp. 446–453.

[22] P. Macko, M. Chiarini, M. Seltzer, and S. Harvard, “Collecting Provenance Via the Xen Hypervisor,” in *The Third USENIX Workshop on the Theory and Practice of Provenance*, 2011.

[23] M. Imran and H. Hlavacs, “Provenance in the Cloud: Why and How?” in *The Third International Conference on Cloud Computing, GRIDs, and Virtualization*, 2012, pp. 106–112.

[24] R. Bose and J. Frew, “Lineage Retrieval for Scientific Data Processing: A Survey,” *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, pp. 1–28, 2005.

[25] Y. L. Simmhan, B. Plale, and D. Gannon, “A Survey of Data Provenance Techniques,” *Computer Science Department, Indiana University, Bloomington IN*, vol. 47405, 2005.

[26] S. B. Davidson, S. C. Boulakia, A. Eyal, B. Ludäscher, T. M. McPhillips, S. Bowers, M. K. Anand, and J. Freire, “Provenance in Scientific Workflow Systems,” *IEEE Data Engineering Bulletin*, vol. 30, no. 4, pp. 44–50, 2007.

[27] C. Goble, “Position Statement: Musings on Provenance, Workflow and (Semantic Web) Annotations for Bioinformatics,” in *Workshop on Data Derivation and Provenance, Chicago*, 2002.

[28] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, “Provenance-Aware Storage Systems,” in *USENIX Annual Technical Conference, General Track*, 2006, pp. 43–56.

[29] C. Sar and P. Cao, “Lineage File System.” [Online]. Available: crypto.stanford.edu/~cao/lineage

[30] R. K. Ko, P. Jagadpramana, and B. S. Lee, “Flogger: A file-Centric Logger for Monitoring File Access and Transfers Within Cloud Computing Environments,” in *The 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom).* IEEE, 2011, pp. 765–771.

[31] A. P. Chapman, H. V. Jagadish, and P. Ramanan, “Efficient Provenance Storage,” in *International Conference on Management of Data.* ACM, 2008, pp. 993–1006.

[32] A. Meliou, W. Gatterbauer, and D. Suciu, “Bringing Provenance to Its Full Potential Using Causal Reasoning.”

[33] S. Gao and C. Zaniolo, “Provenance Management in Databases Under Schema Evolution,” in *The Fourth USENIX Conference on Theory and Practice of Provenance.* USENIX Association, 2012, pp. 11–11.

[34] P. Buneman, J. Cheney, and E. V. Kostylev, “Hierarchical Models of Provenance,” in *The Fourth USENIX Conference on Theory and Practice of Provenance*, 2012, p. 10.

[35] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, “An Annotation Management System for Relational Databases,” *The VLDB Journal*, vol. 14, no. 4, pp. 373–396, 2005.

[36] P. Buneman, S. Khanna, and T. Wang-Chiew, “Why and Where: A Characterization of Data Provenance,” in *Database TheoryICDT 2001*. Springer, 2001, pp. 316–330.

[37] D. Kulkarni, “A Provenance Model for Key-Value Systems,” in *The Fifth USENIX Workshop on The Theory and Practice of Provenance*. USENIX, 2013.

[38] “Google Cloud Data Store.” [Online]. Available: https://developers.google.com/datastore/

[39] “Amazon Simple Storage Service (S3).” [Online]. Available: http://aws.amazon.com/s3/

[40] “Open Stack Software.” [Online]. Available: https://www.openstack.org/

[41] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii, “Solving Big Data Challenges for Enterprise Application Performance Management,” *The VLDB Endowment*, vol. 5, no. 12, pp. 1724–1735, 2012.

[42] “CQL for Cassandra 1.2.” [Online]. Available: http://www.datastax.com/documentation/cql/3.0/cql/aboutCQL.html

[43] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking Cloud Serving Systems with YCSB,” in *The First Symposium on Cloud Computing*. New York, NY, USA: ACM, 2010, pp. 143–154.

[44] “Cassandra Online Documentation.” [Online]. Available: http://www.datastax.com/documentation/cassandra/1.2/cassandra/features/featuresTOC.html

[45] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web,” in *The 29th Annual Symposium on Theory of Computing*. ACM, 1997, pp. 654–663.

[46] B. H. Bloom, “Space/Time Trade-Offs in Hash Coding With Allowable Errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[47] P. Buneman and W.-C. Tan, “Provenance in Databases,” in *The International Conference on Management of Data*. ACM, 2007, pp. 1171–1173.

[48] “Cassandra Metrics.” [Online]. Available: http://wiki.apache.org/cassandra/Metrics

[49] M. Welsh, D. Culler, and E. Brewer, “SEDA: An Architecture for Well-Conditioned, Scalable Internet Services,” in *Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 230–243.

[50] “CASSANDRA-3919 JIRA Issue.” [Online]. Available: https://issues.apache.org/jira/browse/CASSANDRA-3919