Leveraging Metadata in NoSQL Storage Systems

Ala' Alkhaldi, Indranil Gupta, Vaijayanth Raghavan, Mainak Ghosh
Department of Computer Science
University of Illinois, Urbana Champaign
Email: {aalkhal2, indy, vraghvn2, mghosh4}@illinois.edu

***Abstract*—NoSQL systems have grown in popularity for storing big data because these systems offer high availability, i.e., operations with high throughput and low latency. However, metadata in these systems are handled today in ad-hoc ways. We present Wasef, a system that treats metadata in a NoSQL database system, as first-class citizens. Metadata may include information such as: operational history for a database table (e.g., columns), placement information for ranges of keys, and operational logs for data items (key-value pairs). Wasef allows the NoSQL system to store and query this metadata efficiently. We integrate Wasef into Apache Cassandra, one of the most popular key-value stores. We then implement three important use cases in Cassandra: dropping columns in a flexible manner, verifying data durability during migrational operations such as node decommissioning, and maintaining data provenance. Our experimental evaluation uses AWS EC2 instances and YCSB workloads. Our results show that Wasef: i) scales well with the size of the data and the metadata; ii) minimally affects throughput and operation latencies.**

1. Introduction

With the advent of NoSQL stores, large corpuses of data can now be stored in a highly-available manner. Access to this stored data is typically via CRUD operations, i.e., Create, Read, Update, and Delete. NoSQL storage systems provide high throughput and low latency for such operations.

In NoSQL systems such as Apache Cassandra [1], MongoDB [2], Bigtable [3], etc data is organized into tables, somewhat akin to tables in relational databases. For instance Cassandra calls these tables as "column families", while MongoDB calls them as "collections". Each table consists of a set of rows, where each row is a key-value pair or equivalently a data item. Each row is identified by a unique key. Unlike relational databases, NoSQL systems allow schema-free tables so that a data item could have a variable set of columns (i.e., attributes). Access to these data items is allowed via CRUD operations, either using the primary key or other attributes of the data items.

While NoSQL systems are generally more efficient than relational databases, their ease of management remains a challenge as in traditional systems. A system administrator has to grapple with system logs by parsing flat files that store these operations. Hence implementing system features that deal with metadata is cumbersome, time-consuming and results in ad-hoc designs. Data provenance, which keeps track of ownership and derivation of data items, is usually not supported. During infrastructure changes such as node decommissioning, the administrator has to verify by hand the durability of the the data being migrated.

We argue that such information needs to be collected, stored, accessed, and updated in a first-class manner. We call this information as *metadata*. For the purposes of NoSQL systems, we define metadata as essential information about a data item, a table, or the entire storage system, but excluding the data stored in the data items themselves. This includes structural metadata that is relevant to the way tables are organized, administrative metadata used to manage system resources, and descriptive data about individual data items.

We present Wasef [1], a metadata system intended for NoSQL data stores. Wasef functions as a component of the data store and leverages the underlying NoSQL functionalities to deliver its services. Wasef has to address three major challenges. First, it must collect metadata without imposing too much overhead on the foreground CRUD operations arriving from clients. Second, it must allow an administrator or a user to specify (via clean APIs) which metadata is collected and how to use it. Finally, Wasef must scale with: i) the size of the cluster, ii) the size of the data being stored, iii) the rate of incoming CRUD operations, and iv) the size of the metadata itself.

Our work makes the following contributions:

- We present the design and architecture of Wasef, a metadata management system for NoSQL storage systems.
- We implement the W-Cassandra system, a key-value store consisting of Wasef integrated into Apache Cassandra 1.2.
- We implement three important use cases in W-Cassandra:

  1) A flexible implementation of the column drop operation, thus addressing a major JIRA (bug) issue in Cassandra 1.2.
  2) Durability verification for node decommissioning.
  3) A data provenance feature.

This work was supported in part by NSF grant CNS 1319527, AFOSR/AFRL grant FA8750-11-2-0084, and NSF grant CCF 0964471.

1. Arabic word for "Descriptor."

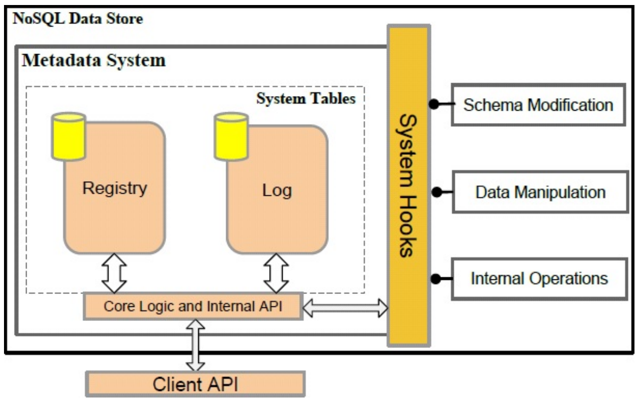


Figure 1: The architecture of Wasef.

- We evaluate W-Cassandra on the AWS cloud [4], and using realistic YCSB [5] workloads. Our evaluation shows that Wasef: i) scales well with the cluster size, data size, operation rate, and metadata size; ii) minimally affects throughput and operational latencies.

The rest of the paper is organized as follows. Section 2 provides an overview of Wasef and its API, and we expand on the details in Section 3. Section 4 describes the design and implementation of the use case scenarios, and evaluation is presented in Section 5. We describe related work in Section 6, and conclude in Section 7.

2. System Design

In this section we first lay out our design principles (Section 2.1). We then describe our architecture (Section 2.2), workflow (Section 2.3) and API (Section 2.4).

2.1. Design Principles

Wasef's design is based on four guiding principles:

1) *Modularity and integration with the existing functionality*: The metadata system should modularly integrate with the underlying infrastructure. It should not affect existing NoSQL APIs, functionality, or performance.
2) *Flexible granularity of collected metadata*: The design should be flexible to collect and store metadata about objects and operations of different kinds and at different granularities (e.g., data items vs. tables). Such metadata includes (but is not fundamentally limited to) the time and description of performed operations, object names, ownership information, and column information.
3) *Accessibility of metadata by internal and external clients*: Metadata needs to be accessible by both external clients (e.g., for data provenance) as well as servers internal to the cluster (e.g., for management operations such as dropping of columns). We provide this via flexible APIs to collect, access, and manipulate metadata.
4) *Minimal collection of the metadata*: Due to the enormous size of data and operations handled by NoSQL data stores, the continuous collection of metadata about every operation might impose a large overhead on the system. To avoid this, Wasef allows the administrator to configure metadata collection for only a selected set of operations.

2.2. Architectural Components

Wasef consists of five major components (Figure 1):

- *Registry*: The Registry is a table for registering objects for whom metadata will be collected. Each Registry entry is identified by two attributes: i) name of the target object (e.g., table, row, or cluster node), ii) name of the operation(s) that will trigger metadata collection about the target object (e.g., table truncation, row insertion, or node decommissioning). NoSQL systems like Cassandra often offer a type of table called "system tables". As these tables are persistent and easily accessible at servers, we store the Registry as a system table.
- *Log*: The Log is a table where collected metadata is stored. Unlike a flat file format, a table-formatted storage allows easy querying. Like the Registry, we store the Log as our second system table.
- *Core Logic and Internal API*: Wasef logic is implemented as a thin wrapper layer around the Registry and Log. To facilitate efficient metadata operations, it is integrated with the underlying NoSQL system (Section 3). Finally, it exposes an API for internal data store components.
- *System Hooks*: The System Hooks component contains implementations dependent on the underlying data store. It monitors data store operations (e.g., schema modification, data manipulation, etc.), and calls the Core Logic to log the metadata into the Log table.
- *Client (External) API*: The client API is a set of functions exposed to external clients (and users) allowing them to register objects and operations for metadata collection.

2.3. Operational Workflow

We give an overview of how metadata is registered, logged, and queried.

Metadata Registration: Metadata collection starts after a metadata target and operation have been registered in the Registry table. Targets are unique names of data entities, such as tables or data items, or internal components of the data store, such as SSTables or cluster nodes. We use a consistent and unified convention for naming metadata targets. Our Cassandra implementation uses the following naming convention:

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

For example, the target name of a column family (table) called `Teacher` that is part of `School` keyspace is named `School.Teacher`, while the target name for a row (data item for a teacher) with key `John` is `School.Teacher.John`.

Operations are names of events occurring to targets, which trigger metadata collection. Examples are schema modification, row insertion, and node decommissioning. For instance, the column drop operation can be registered under the name `AlterColumnFamily_Drop`. Registration can be performed via either the internal or external API.

Metadata Logging: The System Hooks (Section 2.2) continuously monitor system operations. For each operation, the hook contacts the Core Logic, which checks if the operation and its target are in the Registry. If so, the hook collects the relevant metadata for that operation. For instance, for the `AlterColumnFamily_Drop` operation, the metadata collected is the name of the dropped column, the timestamp, and the database session owner. After that, Core Logic logs this metadata in the Log table.

Metadata Querying: The Log table stores the collected metadata. The internal and external clients of the metadata can concurrently query the Log table via the APIs.

2.4. Metadata API

Wasef provides an intuitive CRUD-like API to both developers of the NoSQL system (internal API), and developers of client code (external API).

Internal API: The internal API provides operations for the Registry and the Log tables. The Registry API consists of three functions for managing a target-operation pair:

```
Registry.add(target, operation)
Registry.delete(target, operation)
Registry.query(target, operation)
```

`Registry.add` registers a target-operation pair by adding a new record into the Registry table. `Registry.delete` removes this entry. `Registry. query` returns a boolean indicating if the target-operation pair exists in the Registry.

The API to the Log consists of the following calls:

```
Log.add(target, operation, timestamp, value)
Log.delete/Log.query(
     target, operation, startTime, endTime)
```

`Log.add` inserts a new metadata record into the Log table. It starts by validating the target and operation parameters against the Registry. Then a record is inserted into the Log containing the target-operation pair (as key), operation timestamp, and the metadata about the target-operation in the value field. `Log.delete` removes all metadata for the target-operation pair. The startTime and endTime parameters are optional; if present, Wasef only removes matching records that were timestamped inside that time interval. Finally, `Log.query` returns all the records identified by the mandatory target parameter. The last three parameters are optional and provide flexibility in querying the Log.

External API: Wasef's external API allows external clients to register, unregister, and query metadata:

```
register, unregister, queryAll,queryLatest
Parameters for all four calls:
   (target, operation)
```

The `register` and `unregister` functions are wrappers around `Register.add` and `Register.delete` respectively from the internal API, and provide identical functionality. For convenience we also provide `queryAll` which retrieves all the records identified by the target-operation pair, as well as `queryLatest` which returns only the last inserted metadata record matching the criteria.

3. Implementation of W-Cassandra

We integrated Wasef into Cassandra 1.2. We call the resulting metadata-aware key-value store as W-Cassandra. The code is available for download at:
`http://dprg.cs.uiuc.edu/downloads`.

3.1. Wasef Metadata Storage

Wasef collects and stores metadata by using two types of tables, in a way that offers low read latency and flexible querying. While implementing these techniques, we use underlying Cassandra tables. This enables Wasef to inherit Cassandra's existing functionality such as data compression, caching, fast access, and replication factors.

Concretely, we store all metadata tables as Cassandra's system tables, and collect them in the `system_metadata` system keyspace. Using system tables provides a read-only protection for the metadata schema, and makes it available immediately after the system is bootstrapped.

As explained earlier in Section 2.2, Wasef stores two system tables: the Registry and the Log table. This separation has three advantages: i) because the Registry table is smaller than the Log, it can be queried quickly for each operation during metadata logging (Section 2.3); ii) management operations on registry entries are simplified, and iii) we can cleanly group entries within the Log, e.g., based on metadata insertion time or operation type.

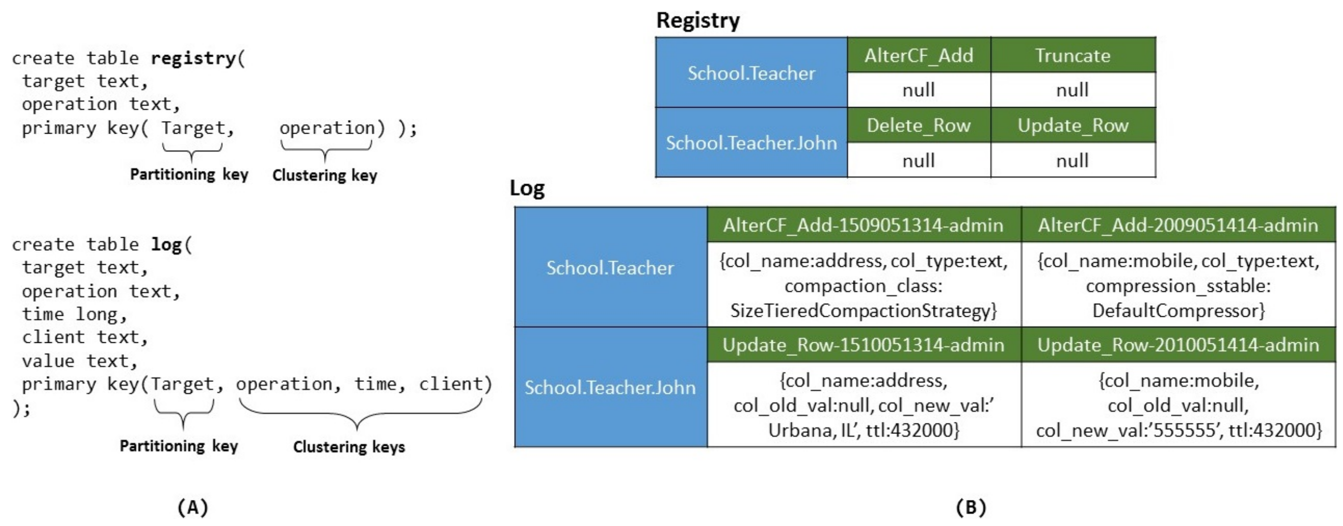
Figure 2 illustrates, via an example, the schemas of the Registry and Log. The Registry table consists of two fields: The `target` field stores the name of the metadata target object, and the `operation` field stores the operation name which will trigger the metadata collection. The Log table has several fields that describe collected metadata. These include the `target`, the `operation`, and the timestamp of the operation (i.e., `time`). The `client` field reports the ownership information of the metadata target.

The primary keys for these tables are carefully chosen to achieve two goals:

1) *Optimizing the storage layout for low read latency:* The `target` key works as the partitioning key for both tables while the clustering keys are joined using a fixed scheme of delimiters. This is shown in Figure 2.B. 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. As shown in Figure 2.B, all metadata for target name `School.Teacher.John` are grouped in the same row in the Log table. Every column in that row represents one operation. Using this layout, performing a select query that asks about all the operations related to one target is as fast as querying about one operation.
2) *Flexible querying of the* `Log` *table:* In CQL, the `where` clause of the `select` statement filters only based on the table primary key. Thus, including more fields in the primary key increases querying flexibility.

3.2. Supported Targets and Operations

Table 1 shows the list of metadata targets and operations currently supported by W-Cassandra: these operations are from among those already present in CQL 1.2 [6].



```
create table registry(
 target text,
 operation text,
 primary key( Target,    operation) );
```

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

Registry

School.Teacher	AlterCF_Add	Truncate
	null	null
School.Teacher.John	Delete_Row	Update_Row
	null	null

Log

School.Teacher	AlterCF_Add-1509051314-admin	AlterCF_Add-2009051414-admin
	{col_name:address, col_type:text, compaction_class: SizeTieredCompactionStrategy}	{col_name:mobile, col_type:text, compression_sstable: DefaultCompressor}
School.Teacher.John	Update_Row-1510051314-admin	Update_Row-2010051414-admin
	{col_name:address, col_old_val:null, col_new_val:' Urbana, IL', ttl:432000}	{col_name:mobile, col_old_val:null, col_new_val:'555555', ttl:432000}

(A) (B)

Figure 2: Storage layout of Wasef. (A) Schemas of metadata tables, described in CQL. (B) Example of the internal storage layout of the metadata tables. Note how column names of the Log table are composed of the clustering primary key names.

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	`Nodetool decommission`	Token ranges

Table 1: Supported metadata targets and operations in W-Cassandra.

Metadata reporting starts after the system is bootstrapped; concretely, after the authentication feature in Cassandra is started. We modified the write path to propagate the ownership information to the metadata Log. The ownership information is needed to implement our data provenance use case (discussed later).

We observe that Table 1 is missing the create operation because we require an explicit `Log.add` to start metadata collection about a target. Implicit creates would have been complicated since it would have required adding non-existing objects in the Registry.

3.3. Optimizing Metadata Collection

Each incoming operation is validated against the metadata Registry. This is the sole overhead entailed for operations that do not have a corresponding registry entry. In case of a matching registry entry, appropriate writes are entered into the Log. To address the overhead of metadata collection for fine-grained metadata targets such as writes for a data item, we optimize both registry validation and log writing:

Fast Registry Validation: We speed up the response time for querying the Registry table in three ways. Each of these leverages underlying Cassandra functionalities.

1) Enabling Dynamic Snitching: We enable dynamic snitching, which allows the Cassandra coordinator to send read requests to replicas that are the closest in round-trip time from the coordinator based on history.
2) Setting read consistency level to `ANY` for the Registry table: This consistency level is faster than `ONE`, and it allows the coordinator to acknowledge the client after either storing it locally or receiving the first replica acknowledgement, whichever occurs earlier. This also reduces the network traffic.
3) Enabling row caching: When row caching is enabled, Cassandra stores new entries in a cache associated with the destination table. Thus, Cassandra can serve read operations from the cache to shorten the read path.

Lightweight Log Writing: We employ two optimization techniques to reduce the Log writing overhead:

1) We set the write consistency level to `ANY` by default, just like for the Registry tables. This improves the time to write an entry into the Log, which is the most common metadata-related operation in W-Cassandra. Depending on the application, one can choose a higher consistency level like, `QUORUM` .
2) We perform write operations in a background thread.

Cassandra uses the SEDA design [7], which divides the work into stages with a separate thread pool per stage. We adopt the same philosophy in order to improve the metadata write efficiency. We do so by injecting write mutations [2] into the mutation handling stage, in a separate thread.

3.4. Discussion

Replication, Fault-tolerance, Consistency Levels: Since Wasef relies on Cassandra's replication (replication factor = 3), Wasef's fault-tolerance guarantees for metadata are identical to Cassandra's fault-tolerance for data. Further, the replication factor is configurable per table in Wasef.

Orthogonal to replication is the issue of consistency level for operations. Wasef writes prefer a consistency level of `ANY` for speed, but this choice does not affect fault-tolerance: a metadata write is acknowledged only after some server has written it to its commit log, thus making it durable. Consistency level only delays propagation of writes to replicas – Cassandra's hinted handoff and read repair mechanisms propagate writes eventually (and quickly) to replicas. Finally, the administrator retains the option to increase Wasef's consistency level to `Quorum` or `ALL`, depending on the application.

Wasef vs. Alternative Approaches: There are two alternative approaches to implementing metadata: i) a standalone implementation running on dedicated servers, and ii) application-specific metadata. Firstly, if metadata were stored on dedicated servers, we would have to re-implement querying, replication, fault-tolerance, and load-balancing. Instead Wasef's integration into Cassandra allows us to directly use these features from Cassandra. Further, dedicated servers would need tuned scaling as datasize and cluster size increases – Wasef scales automatically (as our experiments show). Secondly, implementing metadata in the application would take many man-hours for each application. Our approach is fast yet flexible. If an application does not need metadata, it can disable Wasef.

4. Use Cases: Leveraging The Metadata

Wasef can be leveraged to address system issues and provide new features. To demonstrate this, we implemented three use case scenarios selected from three different domains.

4.1. Enabling Flexible Column Drops

Problem: The JIRA issue 3919 [8] reports that in Cassandra 1.2 dropping a column removes its definition from the table schema but keeps the column data stored in the system. This leads to incorrect behavior, e.g., if another column is added to the table with the same name, CQL queries referencing the new column will still return data belonging to the old column.

Metadata Solution: We present a correct and flexible column drop implementation using Wasef. The intuition behind

2. Cassandra encodes all row-level operations internally in a unified form called Row Mutation.

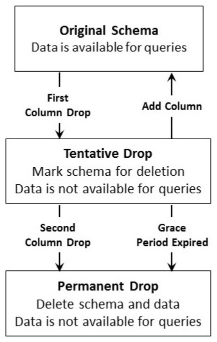


Figure 3: State Machine for Flexible Column Drop in W-Cassandra.

the flexibility in our approach is as follows – in today's file systems (e.g., Mac, Windows, Unix) when a user deletes a file or folder, the user has a second chance to retrieve it from a Trash/Deleted Items folder. Concretely, Figure 3 illustrates the state diagram for flexible column drop. When a column is dropped the first time, it becomes unavailable for CQL queries, but the data is retained. Thereafter, either on a user's explicit second delete command or after a (configurable) timeout, the column data is permanently purged. The user can un-delete the column from the tentatively dropped state.

The state diagram is implemented as follows: when a column is first dropped using the `Alter Table` operation, W-Cassandra inserts a metadata called `AlterColumnFamily_Drop` into the Registry. If the column is subsequently re-added, then this metadata is deleted, leaving the system in the initial state. However, if the column were to be dropped again explicitly or the timeout expires, a metadata log entry with the column name and the time of the drop operation is inserted into the Log table and the column is marked for permanent deletion. Before permanent deletion, any attempts to insert new columns with the old name are rejected. During a CQL select query, W-Cassandra checks both the `AlterColumnFamily_Drop` metadata and the log entry, and excludes the dropped (tentative or permanent) columns from the query's results.

In fact, Cassandra provided a fix for the column-drop issue in a later release (Cassandra 2.0) by retaining a history of dropped columns. However, this history is maintained in a specialized hash map attached to the column schema, and the hash map is replicated at *all* cluster nodes. This ad-hoc solution took many man-months to implement. Instead, our W-Cassandra approach took only 50 additional lines of code to write, is more flexible and systematic, and leverages underlying NoSQL features such as replication.

4.2. Node Decommissioning

Problem: When a node is decommissioned in Cassandra (using NodeTool), its token ranges need to be reassigned. This is a major pain point [9] today as the admin needs to manually count token (i.e., key ranges) and check that none is lost.

Solution: Using our metadata system we can implement an automated way of decommissioning a node. First, the decommissioned node streams its data to other nodes in the cluster and then unbootstraps. Second, the token ranges, at each server, for each non-system table are collected. Third, the partitioner and the replication strategy are used to decide the new replica for each token range. Fourth, all the collected information is passed to the file streamer to move the blocks (SSTables) to the intended destinations. Finally, the decommissioned node is retired.

During the decommission command, we add new replica for token ranges from the decommissioned node into the metadata Log, by using a metadata tag `decommission`. Thereafter we automatically call NodeTool to verify the decommissioning operation's correctness. NodeTool fetches metadata from Wasef and checks that all token ranges are accounted for.

4.3. Providing Data Provenance

Problem: Data provenance captures where data comes from, when it was produced, etc. Many disciplines like, astronomy, bio-informatics, etc rely on data provenance. Unfortunately, modern NoSQL systems largely do not support data provenance collection.

Solution: Using Wasef (Section 3.2), we collect the following provenance data about each operation in Cassandra:

1) Full name of target of data operation. E.g., dropping a table called `User` located in `Test` keyspace results in logging `Test.User` as the full name.
2) Operation name. E.g., `Alter_Add_Column Family` indicates a new column addition to a column family.
3) Operation time: timestamp of the operation.
4) The authenticated session owner name.
5) Results of the operation. E.g., when a new column is added, the name of the column and its attributes. When a column name is modified, its old and new names.

This provides: i) *Where Provenance*, which keeps track of the records from which the query results were derived. In the absence of joins in NoSQL systems, Wasef provides Where Provenance through the "full name" of the metadata target. ii) *Why Provenance*, the justification of the results via a listing of operations that produced them in Wasef's "operation name" field in the Log.

This provenance data can be queried via our external APIs (Section 2.4). Features such as replication, scalability, and accessibility are enabled as usual for provenance data. For correctness, we do not automatically garbage-collect old provenance data, but deletes can be done manually by system administrators, e.g., based on their timestamps.

5. Experimental Evaluation

We answer the following questions:

1) What is the performance cost of integrating metadata collection and querying into Cassandra? This includes read and write latencies, and the overall throughput, for W-Cassandra.
2) How does W-Cassandra scale with cluster size, size of data, size of metadata, and query injection rate?
3) How does W-Cassandra perform for the use case scenarios of Section 4?

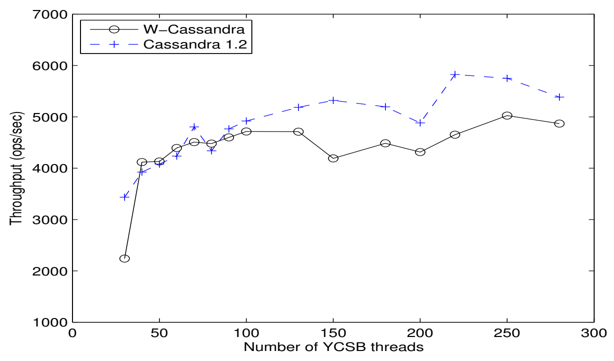


Figure 4: **Evaluating throughput against number of clients. Each point in the graph is the average ops/s arising from a workload of 1 million YCSB client requests.**

We run our experiments on the Amazon Web Services (AWS) EC2 public cloud infrastructure [4]. We inject traces using the YCSB benchmark [5].

5.1. Experimental Setup

Our experiments use six AWS EC2 m1.large instances, each with 2 virtual CPUs (4 ECUs), 7.5 GB of RAM, and 480 GB of ephemeral disk storage. We run the YCSB client from a separate identical instance. The instances use Ubuntu 12.04 64-bit operating system with swapping turned off, as recommended by Cassandra for production settings. We use zipfian distribution to generate YCSB workloads, i.e. some popular keys are accessed more often than others.

5.2. W-Cassandra Throughput and Latency

We first measure the effect of Wasef on Cassandra's throughput and latency. We conduct 15 YCSB runs for both W-Cassandra and standard Cassandra (1.2) using a workload of 50% update and 50% read. The database size is 12 GB, keys are 8 B, and values are 1 KB. The metadata log starts of as empty and each update is logged as metadata.

Figure 4 shows the throughput, which increases quickly at first and then saturates at about 300 threads. Over all the data points in the plot, the average performance of W-Cassandra is only 9% worse than Cassandra. This value captures the overhead of Wasef's metadata collection.

Figure 5 depict the update latencies for the two systems. Over all the data points in the plot, the average update latency is 15% worse. This is because an update engenders additional metadata writes. For read latency (plot in tech report [10]), the overhead is 3%.

5.3. Scalability with Cluster Size

We linearly increase the number of nodes in the cluster, while proportionally scaling the data set size and system load. From one run to the next in Figure 6, the cluster size was increased by two nodes, the data set increased by 4 GB, and load increased by 50 YCSB threads.

Figure 6 shows that W-Cassandra retains linear scalability in spite of its metadata overheads ("Scalability difference" line). The percentage overhead in update latency rises

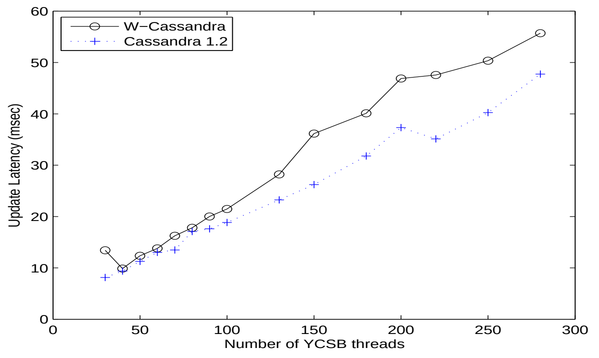


Figure 5: **Update latency against number of clients.**

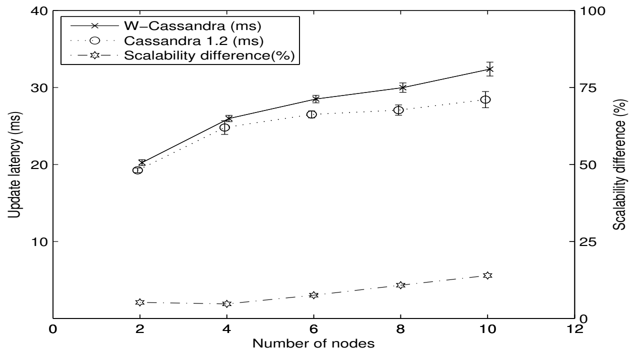


Figure 6: **Scalability against cluster size. Experiment increases the number of cluster size by 2 nodes, data size by 4 GB, and YCSB load by 50 threads. Datapoints are perturbed horizontally for clarity.**

slowly with scale because W-Cassandra injects one metadata validation request (i.e., read request) per update. With high probability, this request is served locally when the number of nodes is small. However, when the number of nodes increase this probability decreases. Yet, the overhead is still small: at 10 nodes, the update latency increase due to W-Cassandra is only about 10%.

5.4. Column Drop Feature

We compare the latency for column drop in W-Cassandra with that in Cassandra 1.2 using a data set size of 8 GB. While Cassandra 1.2 implements a crude (and incorrect [3]) version of column drop, we choose to compare against it because: i) Wasef is built into Cassandra 1.2, and ii) comparing against the latest version Cassandra 2.0 would be unfair as this version is faster than 1.2 because of several optimizations that are orthogonal to column dropping.

Since the YCSB benchmark does not offer schema modification tests, we designed a customized test that performs a set of 500 column drop operations. Figure 7 shows that the latency of W-Cassandra hovers at or around Cassandra's column drop latency. Since Cassandra 1.2's implementation is incorrect, this is a positive result.

3. Cassandra 1.2 's column drop deletes the schema definition but retains the data.

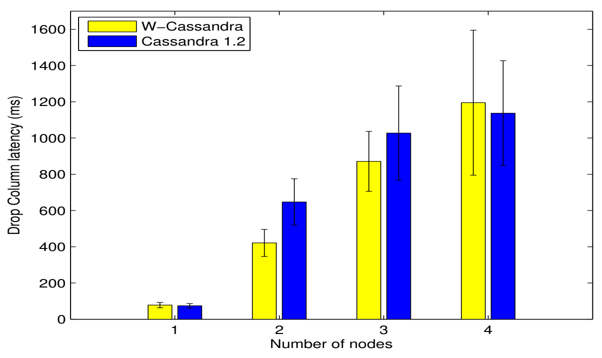


Figure 7: **Column drop latency against cluster size. Each bar in the graph is average of 500 drop column operations performed by clients running at a separate machine.**

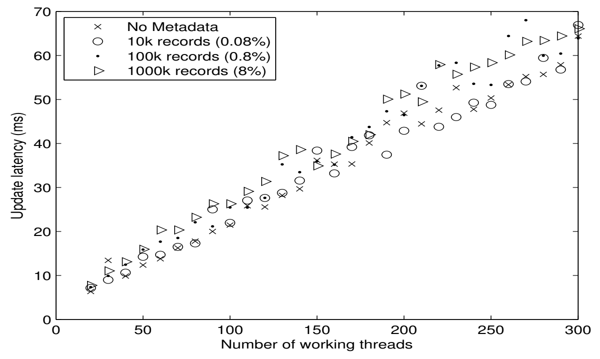


Figure 8: **Update latency comparison between different metadata sizes registered in W-Cassandra.**

5.5. Scalability with Data Size

Figure 8 shows that as the metadata size is increased from 0.08% to 8% of data size, the increase in update latencies, while provenance is being collected, is generally very small. The observation is similar for read latencies [10]. Independent of its size, this metadata is in fact replicated across multiple servers, thus allowing it to scale with data size. Finally, we note that Wasef is memory-bound rather than disk-bound because Cassandra is too. A disk-bound Cassandra would be very slow, and would lead the administrator to add more servers, making it, and thus Wasef, memory-bound again.

5.6. Verifying Node Decommissioning

The main overhead faced by the system administrator during node decommissioning is the first stage when token metadata is collected; thereafter the data streaming to other servers is automated. To measure the overhead of the first stage, we vary the number of tokens per node. We use four AWS EC2 instances, and a 4 GB data set size. Figure 9 shows that W-Cassandra is only marginally slower than Cassandra; the average overhead was measured at 1.5%.

6. Related Work

Metadata: Metadata systems can be implemented internal to a database [11] or externally [12]. Examples of external

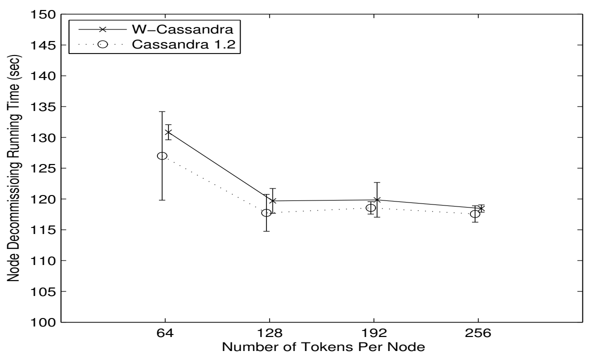


Figure 9: **Running time for node decommissioning operation. Datapoints are perturbed horizontally for clarity.**

metadata systems include those for businesses [13] and for Grids [14]. Internal metadata systems [3] are used to collect structural metadata.

Many have argued that metadata should be a feature of cloud data stores [15], [16]. Client-centric approaches for metadata [15] are too intrusive; we believe that metadata collection should be server-centric.

Data Provenance: Provenance information is managed in scientific workflows [17], monitoring system operations [18], and database queries [19], [20]. Query provenance in relational database is of two kinds [21]: 1) *Where Provenance* describes source records of a query's result, and 2) *Why Provenance* justifies query results by its source operations and relations between source records. Wasef provides both these kinds of provenance.

There has been some recent work on provenance in key-value stores [11], [15], [22]. The KVPMC system for Cassandra [22] collects provenance data on request, provides client access, and can store provenance data internally or externally. However, Wasef is preferable for four reasons: 1) it is a general solution for any modern NoSQL system, 2) it collects all kinds of metadata, not merely provenance, 3) KVPMC is client-side while Wasef is server-side, and 4) Wasef imposes less overhead and provides good scalability.

7. Conclusion

We presented a metadata system for NoSQL data stores, called Wasef. We integrated Wasef into Cassandra. We showed how our system, called W-Cassandra, can be used to correctly and flexibly provide features like column drop, node decommissioning, and data provenance. Our experiments showed that our system imposes low overhead on Cassandra throughput of 9% and update latency of 15%. We also showed that our system scales well with cluster size, incoming workload, data size, and metadata size. We believe that Wasef opens the door to treating metadata as first-class citizens in NoSQL systems, and exploring the myriad forms of metadata that abide in this new class of data stores.

References

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

[2] "MongoDB," https://www.mongodb.org/.

[3] 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.

[4] "Amazon Web Services (AWS)," http://aws.amazon.com/.

[5] 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.

[6] "CQL for Cassandra 1.2," http://www.datastax.com/documentation/cql/3.0/cql/aboutCQL.html.

[7] 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.

[8] "CASSANDRA-3919 JIRA Issue," https://issues.apache.org/jira/browse/CASSANDRA-3919.

[9] "Slightly Remarkable Blog, Removing Nodes from a Cassandra Ring," http://slightlyremarkable.com/post/57852577144/removing-nodes-from-a-cassandra-ring.

[10] A. Alkhaldi, I. Gupta, V. Raghavan, and M. Ghosh, "Leveraging metadata in nosql storage systems," University of Illinois, Urbana-Champaign, Tech. Rep., 2015, http://hdl.handle.net/2142/73262.

[11] 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.

[12] 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.

[13] "Oracle Metadata Service in Fusion Middleware 11g," http://www.oracle.com/technetwork/developer-tools/jdev/metadataservices-fmw-11gr1-130345.pdf.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] P. Buneman, S. Khanna, and T. Wang-Chiew, "Why and Where: A Characterization of Data Provenance," in *Database Theory-ICDT 2001*. Springer, 2001, pp. 316–330.

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