Evaluation of Data Storage in HathiTrust Research Center Using Cassandra

Guangchen Ruan
Data to Insight Center
School of Informatics and Computing
Indiana University
gruan@indiana.edu

Beth Plale
Data to Insight Center
School of Informatics and Computing
Indiana University
plale@indiana.edu

ABSTRACT

As digital data sources grow in number and size, they pose an opportunity for computational investigation by means of text mining, NLP, and other text analysis techniques. The HathiTrust Research Center (HTRC) was recently established to provision for automated analytical techniques on the over 11 million digitized volumes (books) of the HathiTrust digital repository. The HTRC data store that hosts and provisions access to HathiTrust volumes needs to be efficient, fault-tolerant and large-scale. In this paper, we propose three schema designs of Cassandra NoSQL store to represent HathiTrust corpus and perform extensive performance evaluation using simulated workloads. The experimental results demonstrate that encapsulating the whole volume within a single row with regular columns delivers the best overall performance.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Experimentation, Performance, Schema design

Keywords

Cassandra, schema design, performance evaluation

1. INTRODUCTION

HathiTrust Digital Library [10] is a digital preservation repository that provides long-term preservation and access services for public domain and in copyright content from a variety of sources. As of 2014, HathiTrust has digitized just over 11 million volumes (books) from research libraries across the country. The HathiTrust Research Center (HTRC) [11] was recently established to provision for automated analytical techniques on the text data of the HathiTrust digital repository. Due to the scale of the HathiTrust corpus and the stringent performance requirement imposed by HTRC data services (i.e. the caller), it has been very challenging to design a successful data store. The challenges include:

- *Large-scale and Scalability* — HTRC data store needs to handle the TB scale HathiTrust corpus data and to easily scale up to accommodate forthcoming corpus (e.g. copyrighted content).

- *Availability and Fault tolerance* — HTRC is committed to provision production services which require HTRC data store be able to deliver an "always-on" service or degrade elegantly even under the circumstance of nodes failure.

- *Efficiency* — HTRC data store needs deliver efficient reads and writes operations to meet the performance requirement of HTRC RESTful data services, in other words, HTRC data store should satisfy the *service-level agreement* (SLA) contracted with HTRC data services.

Since traditional data store solutions such as pairtree [16] and relational databases [5, 17] have scalability and availability issues, choosing a NoSQL [3] solution becomes straightforward and we use Cassandra, an exemplary and powerful extensible record store to host HathiTrust volumes. However, it is still uncertain which schema should be used to represent the corpus data and what is the expected performance. To answer these questions, in this paper we propose three column family schemas and conduct extensive performance evaluations under various simulated workloads. The experimental results demonstrate that encapsulating the whole volume within a single row with regular columns achieves the best overall performance amongst all three schemas whilst keeping the representation concise and clear. Our evaluation also demonstrates Cassandra's capability of handling massive workload in concurrent access scenario.

The remainder of the paper is organized as follows: Section 2 discusses related work. Section 3 presents the proposed schemas. Sections 4 and 5 present experimental results under concurrent readers/writers access scenario and user query scenario, respectively. Section 6 presents and discuss experimental results on simulated workloads. Finally we conclude the paper in Section 7.

2. RELATED WORK

In this section we discuss the issues of pairtree [16] and relational databases [5, 17] when facing large-scale and size growing data. We present the rationales behind the design of NoSQL data stores and elaborate some exemplary NoSQL solutions.

Pairtree [16] is a file system hierarchy for the organization of a digital object store used by digital libraries. The pairtree algorithm maps an arbitrary UTF-8 encoded object identifier string into

a file system directory path based on successive pairs of characters. An object directory holds all the files that comprise the object. Any file system can be employed as the underlying storage and backup and restore can simply be performed with native file system tools. Pairtree has two major limitations: 1) Pairtree does not scale well when the number of objects is tremendous. The system cost of maintaining the whole tree structure can be prohibitive, e.g., Linux *inodes* may consume more storage space than files themselves when most of the files are small. Retrieval can also be inefficient when accessing deep branches; and 2) since the mapping between the object identifier and the file system directory is rule-based and does not consider the distribution of identifiers, pairtree can be quite imbalanced and therefore no consistent performance can be guaranteed, e.g., like an AVL tree.

Relational databases [5, 17] are very good at solving certain data storage problems, but because of their focus, several issues emerge when it is time to scale. Expensive joins need to be minimized, which means denormalizing the data, which means maintaining multiple copies of data and seriously disrupting the design, both in the database and in the application. Making schema changes without taking shards "offline" is also a challenge. Experience has shown that data stores that provide *ACID* guarantees tend to have poor availability. This has been widely acknowledged by both the industry and academia [8].

To address these issues, quite a few NoSQL systems have been designed and they can be roughly classified into the following three categories: 1) key-value stores, which include Dynamo [8], Voldemort [22], Scalaris [19], and Riak [18]; 2) document stores, which include CouchDB [7], MongoDB [15], SimpleDB [20], and Terrastore [21]; and 3) extensible record stores, which include BigTable [4], Cassandra [14], HBase [12], HyperTable [13], and PNUTS [6]. These NoSQL systems are designed to be distributed, decentralized, elastically scalable, highly available, fault-tolerant, and tunably consistent. They restrict the traditional notion of ACID transactions by allowing only single-record operations to be transactions and/or by relaxing ACID semantics, for instance, supporting only "eventual consistency" on multiple versions of data. Below we elaborate the design and implementation details of some of them.

Bigtable [4] is a distributed storage system for managing structured data that is designed to scale to a very large size. A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes. Bigtable is built on several other pieces of Google infrastructure. Bigtable uses the distributed Google File System (GFS) [9] to store log and data files and relies on a highly-available and persistent distributed lock service called Chubby [2].

Dynamo [8] is a highly available key-value storage system that is used in some of Amazon's core services to provide an "always-on" experience. Dynamo employs consistent hashing as the partition scheme to distribute the load across multiple storage hosts. Nodes are logically organized in a ring. A gossip based protocol propagates membership changes and helps every node maintain information about every other node. In this manner, request routing can be done with at most one-hop. Dynamo detects updated conflicts using a vector clock scheme, but prefers a client side conflict resolution mechanism. Dynamo is quite flexible to meet different performance requirement. Service owners can customize their storage system to meet their desired performance, durability and consistency SLAs simply by tuning the corresponding parameters.

Cassandra [14] is an open source distributed database management system. It is designed to handle very large amounts of data spread out across many commodity servers while providing a highly available service with no single point of failure. Cassandra is a highly scalable, eventually consistent, distributed, and structured extensible-record store, which brings together the distributed systems technologies from Dynamo and the data model from Google's BigTable. Like Dynamo, Cassandra is eventually consistent. Like BigTable, Cassandra provides a column family-based data model richer than typical key/value systems.

3. COLUMN FAMILY DESIGN

The original volumes are stored in pairtree in HathiTrust's filesystem where each volume corresponds to a directory. Under each directory there is a *mets* xml file which contains all metadata information about the volume and a zipped file which contains the raw .txt files for individual pages. Before ingesting data into Cassandra, a schema needs to be designed to represent the volume. There are two sorts of data, i.e., *page level* data and *volume level* data; and the principle we adopt is to group logically related fields together. In Cassandra, there are two sorts of columns, i.e., *regular column* and *super column*. Regular column can be simply viewed as a mapping between the column name and the column value; while a super column is a collection of sub-columns/regular-columns. Therefore several choices can be made on how to organize the information together. Based on aforementioned principle and Cassandra's data model, we propose three column family designs which differ from each other in terms of the granularity and the compactness in data representation.

Cassandra is actually schema-less and for convenience in the following sections we still use the term "schema" in relational database field to refer to the design of column families. We define *keyspace* (analogous to *database* in RDBMS) *"HTRCCorpus"*, within which three column families (analogous to *table* in RDBMS), i.e., *"VolumeContents"*, *"Collections"* and *"CollectionNames"* are defined. The *"Collections"* column family keeps track of copyright information. It consists of two rows which record IDs of public domain and in-copyright domain volumes, respectively. As in Cassandra there is no easy or inexpensive means to list all row keys within a column family, *"CollectionNames"* is employed as a dedicated column family to record all volume IDs. It is composed of a single row whose column names are volume IDs and column values are irrelevant and only serve as place holders. The *"VolumeContents"* is the core column family which carries the real "payload", i.e. storing the actually volume metadata and content. In below we present three proposed schemas for *"VolumeContents"* column family.

3.1 Encapsulating the Whole Volume within a Single Row with Super Columns

In this schema, each volume is stored within a single row in *"VolumeContents"*, which in turn consists of multiple super columns; one super column for metadata and one super column for each page (referred to as *schema 1* henceforth). The schema of each column family is listed in Fig. 1.

3.2 Encapsulating the Whole Volume within a Single Row with Regular Columns

In contrast with *schema 1* which groups related information into a super column, in this design *"VolumeContents"* uses only stan-

```
--keyspace
"HTRCCorpus" {

  --the master super column family, which
  contains all volume and page contents and
  metadata
  "VolumeContent" {

  --row key, which is the volume ID
  "volumeID" {

  --super column key for volume metadata
  "metadata" {

  --sub column for volume copyright info
  "copyright"; <PUBLIC | IN-COPYRIGHT>

  --sub column for number of pages in the volume
  "pageCount"; <page_count>

  --sub column for publish date of volume
  "publishDate"; <publishDate of volume>

  --sub column for genre of volume
  "genre"; <genre of volume>

  --sub column for zip file of volume
  "wholeVolume"; <zip file of volume> }

  --super column key for the first page
  "00000001" {

  --sub column for page contents
  "contents"; <page_text>

  --sub column for number of bytes in page
  contents
  "byteCount"; <byte_count> }
  } }

  --column family for themed collections
  "Collections" {

  --row containing public volumes
  "public" {

  --valueless column where column key is the
  volume ID
  "volumeID" }

  --row containing in-copyright volumes
  "in-copyright" {

  --valueless column where column key is the
  volume ID
  "volumeID" }
  }

  --column family with one row listing all
  available collections
  "CollectionNames" {

  "name" {

  --valueless column where column key is a named
  collection
  "collectionName" }
  }
}
```

Figure 1: Schemas of three column families within the *"HTR-CCorpus"* keyspace. *"VolumeContent"* column family encapsulates the whole volume within a single row with super columns. *"Collections"* column family keeps track of copyright information and *"CollectionNames"* column family is employed to record all volume IDs. Blue colored lines are comments.

dard or regular columns to organization information (referred to as *schema 2* henceforth). In *schema 2* basically sub columns in *schema 1* are extracted from super columns and are treated as regular columns, and each volume still corresponds to a single row within the column family. Naming in *schema 2* is simple and straightforward, i.e., the names of the belonging sub column and the super column in *schema 1* are concatenated to name the regular column. The schema of *"VolumeContents"* is shown in Fig. 2.

```
"HTRCCorpus" {

  "VolumeContent" {

  --row key, which is the volume ID
  "volumeID" {

  --info for metadata

  --standard column for volume copyright info
  "metadata.copyright"; <PUBLIC | IN-COPYRIGHT>

  --standard column for number of pages in the
  volume
  "metadata.pageCount"; <page_count>

  --standard column for publish date of volume
  "metadata.publishDate"; <publishDate of volume>

  --standard column for genre of volume
  "metadata.genre"; <genre of volume>

  --standard column for zip file of volume
  "metadata.wholeVolume"; <zip file of volume>

  --info for the first page

  --standard column for page contents
  "00000001.contents"; <page_text>

  --standard column for number of bytes in page
  contents
  "00000001.byteCount"; <byte_count> }

  }
}
```

Figure 2: Schema of *"VolumeContent"* column family which encapsulates the whole volume within a single row with regular columns.

3.3 One Row for Metadata and One Row for Each Page with Regular Columns

In this design, instead of using a single row to encapsulate all volume information (i.e. metadata and pages), metadata and each page correspond to a single row composed of regular columns. Virtually each super column in *schema 1* is represented as a separate row in this schema (referred to as *schema 3* henceforth). Naming for regular columns in *schema 3* is done by simply using sub column names in *schema 1*. To name rows, we concatenate the original row name (i.e. the volume id) with the name of the super column. The schema of *VolumeContents* under this design is shown in Fig. 3.

Since all three proposed schemas separate the content (or metadata) for different pages into different super/regular columns, they allow fast locating and retrieving when the user only interests in a particular set of pages such as preface or table of contents. However, the drawback is that when the user needs the whole volume we have to retrieve each page one by one, which is not quite efficient. The workaround is that compressing all pages and metadata files into a single zipped file and store that file in a separate column in Cassandra. In such a means, the whole volume can be obtained by accessing only one sub/regular column (i.e. the *"wholeVolume"* column). Here, we trade space for access time and the cost of adding extra storage is a small price to pay for the performance.

4. EXPERIMENTAL RESULTS OF CONCURRENT READS/WRITES

Apache Cassandra is a free, open source and distributed data storage system that differs sharply from relational database management systems. It evolves rapidly and makes minor releases from time to time. As we started this work, the most stable version is 0.8.2, and later Cassandra releases version 1.0.1 with many optimization and enhanced features. Therefore we switch to version 1.0.1 at that point and all subsequent performance evaluations are conducted based on that version. A 3-node Cassandra cluster is set up for performance evaluation. Each node is a commodity workstation with following specifications: 2 Intel (R) Xeon (R) cores of 2.40 GHz, 6 GB main memory and 460 GB hard disk. The Cassandra cluster is configured with replication factor of two and each node hosts equal share of data. Each cassandra instance uses 25% available memory as JVM heap, which is 1.5 GB. Two corpora are used for evaluation: a medium size corpus—Indiana University collection which contains 28,896 volumes and a large size corpus—nongoogle collection which contains 256,451 volumes.

```
"HTRCCorpus" {

    "VolumeContent" {

    --row key, which is the metadata for volum with
    ID "volume ID"
    "volumeID.metadata" {

    --info for metadata

    --standard column for volume copyright info
    "copyright"; <PUBLIC | IN-COPYRIGHT>

    --standard column for number of pages in the
    volume
    "pageCount"; <page_count>

    --standard column for publish date of volume
    "publishDate"; <publishDate of volume>

    --standard column for genre of volume
    "genre"; <genre of volume>

    --standard column for zip file of volume
    "wholeVolume"; <zip file of volume> }

    --info for the first page
    --row key, the first page of volume with ID
    "volume ID"
    "volumeID.00000001" {

    --standard column for page contents
    "contents"; <page_text>

    --standard column for number of bytes in page
    contents
    "byteCount"; <byte_count> }

    }
}
```

Figure 3: Schema of *"VolumeContent"* column family which uses one row for metadata and one row for each page with regular columns.

Concurrent readers/writers experiments simulate the read/write conflict in real world applications. There are three scenarios, (1) multiple readers only; (2) multiple writers only; and (3) both readers and writers. For each run, firstly a subset is randomly selected from the whole corpus. Each reader/writer then randomly accesses a portion of volumes from this subset in the following means: readers read the whole record of the volume (all metadata and content of each page); and writers update the volume with the same content, in other words, the whole record of the volume is simply rewritten without any change. The subset is chosen as 5% of the whole corpus and readers/writers randomly access 60% of the subset, which is roughly 3% of the corpus (5% * 60% = 3%) or 867/7694 (IU/nongoogle) volumes. Under this setting two clients will have at least 20% overlap on accessed volumes (i.e. the conflict). As the subset is selected randomly and in order to obtain reliable results, the results showed in following figures for each condition are actually the averaged results over 5 independent runs.

Figure 4(a) shows the performance comparison of three schemas on IU collection under multiple readers case. Each reader reads 887 volumes (3% of the corpus) from the same subset. X-axis shows the number of readers and Y-axis shows the time consumed to read the 887 volumes. We can observe from Fig. 4(a) that *schemas 1* and *2* provision much more efficient read operation than *schema 3*. In order to retrieve the content of a volume, *schema 3* has to invoke one row access for each page while *schemas 1* and *2* only need one. Excessive row accesses greatly degrade the read performance of *schema 3*. *Schema 1* performs slightly better than *schema 2* and our explanation is that *schema 1* has less *sorting cost* compared with *schema 2*: Cassandra executes a sort operation on results before returning them back to the client. In *schema 1*, sort is executed on super column level, which is much cheaper than *schema 2*, where the sort has to be conducted on regular columns whose number is much greater than that of super columns in *schema 1*. Another observation we have is that Cassandra shows good linear scalability, i.e., there is no noticeable performance degradation as the number of readers increases.

Figure 4(b) shows the performance comparison under multiple writers case. Each writer writes 887 volumes (3% of the corpus) from the same subset. X-axis shows the number of writers and Y-axis shows the time consumed to write the 887 volumes. We can observe from Fig. 4(b) that *schema 2* is the best one in terms of write operation. *Schema 3* is still the worst one because of too many row accesses. The performance of *schema 1* is inferior to *schema 2* in that composing super columns is more expensive than regular columns. We also observe linear scalability in this concurrent writes scenario.

Figure 4(c) shows the performance comparison under multiple readers / writers case. Each reader (writer) reads (writes) 887 volumes (3% of the corpus) from the same subset. X-axis shows the number of readers (writers) and Y-axis shows the time consumed to read (write) the 887 volumes. The results are consistent with those observed in Figures 4(a) and 4(b). *Schema 2* achieves the best overall performance amongst all three in terms of both reads and writes.

To further evaluate the performance of *schema 2*, we scale to the larger nongoogle collection and repeat the above three cases. Figure 5(a) shows the performance of *schema 2* on nongoogle collection under multiple readers case. Each reader reads 7,694 volumes (3% of the corpus) from the same subset. X-axis shows the number of readers and Y-axis shows the time consumed to read the 7,694 volumes. Although the size of the volumes (7,694) being read is roughly 8.7 multiples of the IU collection case (887), the time consumed (66s) is only 4.7 multiples of the IU collection case (14s), which demonstrates another dimension of linearity.

Figure 5(b) shows the multiple writers case. Each writer writes 7,694 volumes (3% of the corpus) from the same subset. X-axis shows the number of writers and Y-axis shows the time consumed to write the 7,694 volumes. As the size of the data becomes large, we observe more outliers than in the IU collection case. However,
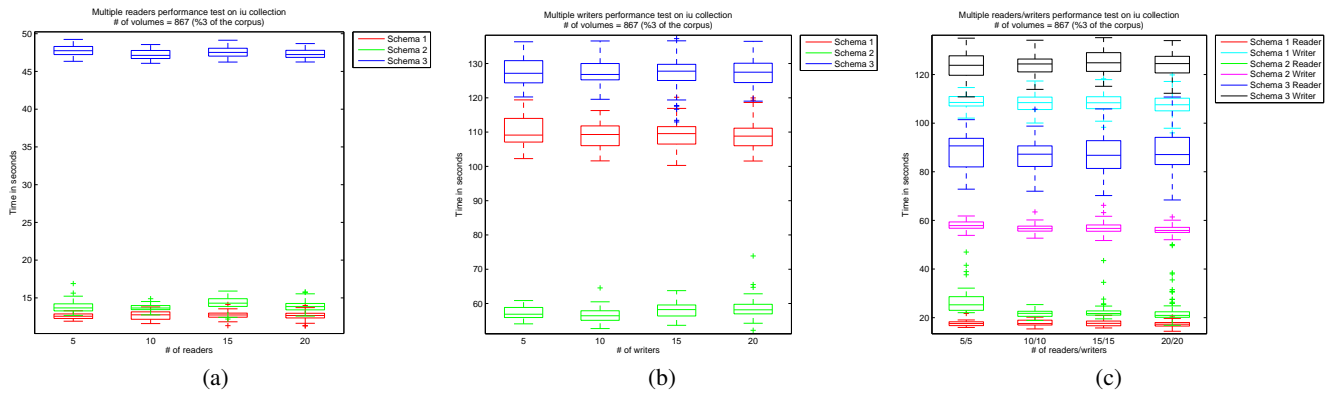
Figure 4: Performance comparison of three schemas under concurrent readers and/or writers access, IU collection used. (a) readers only, (b) writers only, and (c) both readers and writers.
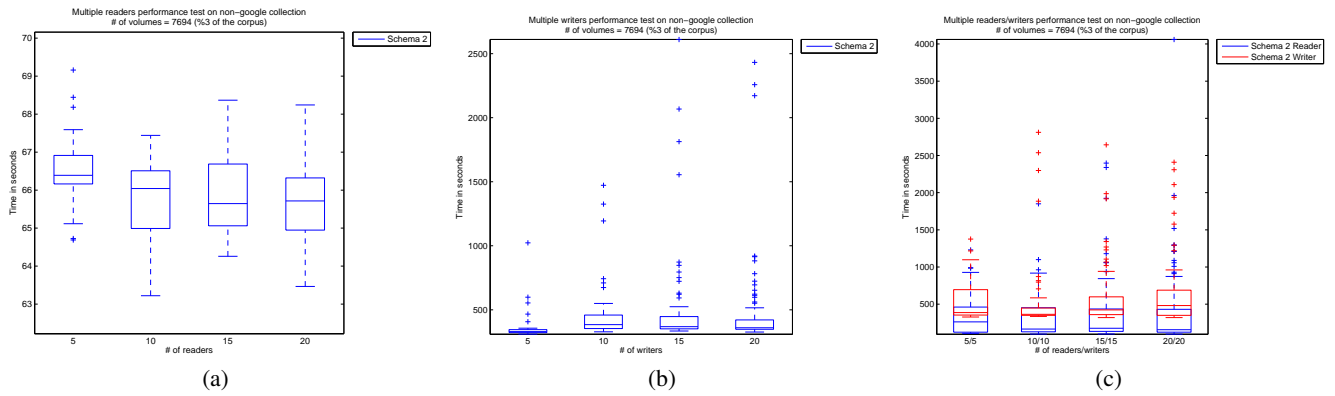

Figure 5: Performance of schema 2 under concurrent readers and/or writers access, nongoogle collection used. (a) readers only, (b) writers only, and (c) both readers and writers.
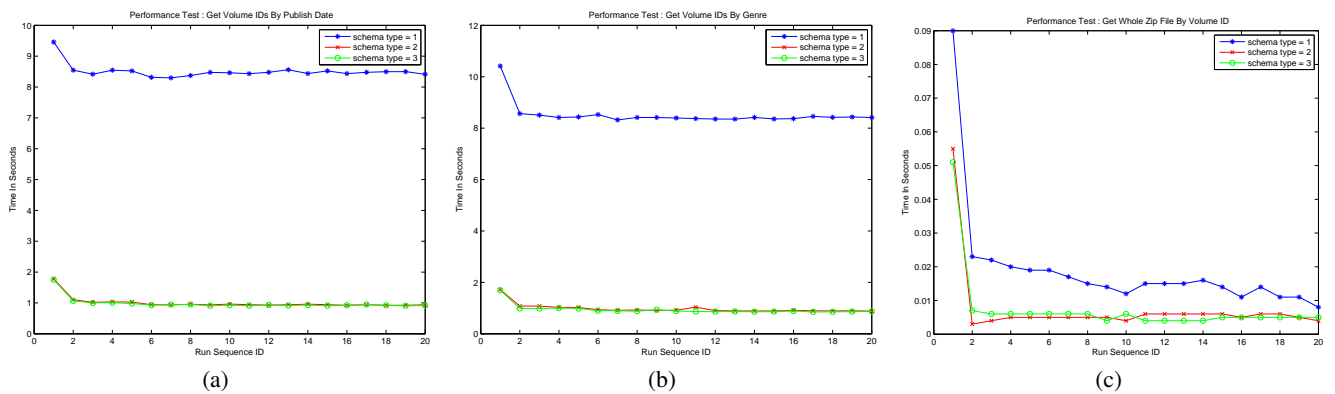

Figure 6: Performance comparison of three schemas under three user queries. (a) retrieve volume IDs by publish date, (b) retrieve volume IDs by genre, and (c) retrieve the whole volume content by volume ID.

as the box is actually the averaged results over 5 independent runs, outliers only account for a small portion.

Figure 5(c) shows the multiple readers/writers case. Each reader (writer) reads (writes) 7,694 volumes (3% of the corpus) from the same subset. X-axis shows the number of readers (writers) and Y-axis shows the time consumed to read (write) the 7,694 volumes. We see more outliers when the number of readers and writers increases. The time consumed for read (write) operations, however, doesn't increase linearly as the number of readers and writers, which is a desired scalability property.

Discussion. From the concurrent readers/writers experiments we can observe that *schema 2* achieves the best overall performance amongst all three proposed schemas. Below we discuss pros and cons of each schema in detail. Encapsulating a whole volume within a single row with super columns (*schema 1*) is quite straightforward and has a better hierarchical structure, i.e., one super column for metadata and one super column for each page. Related information is grouped together within one super column which consists of multiple regular columns. However, decomposing super column is a costly operation and thus accessing sub-columns within super columns is far less efficient than directly accessing regular columns. *Schema 1* achieves compact and concise representation at the expense of performance. In *schema 3* we employ one row for metadata and one row for each page, all with regular columns. Under this design, there is no "fat" row as each row only represents a limited amount of information. Accessing a single row becomes very fast and this property can be quite beneficial when clients only need to retrieve a small number of rows. For instance, clients only care about metadata or a particular set of pages such as preface or table of contents. However, when dealing with cases that clients need to conduct analysis over the whole content of the volume, this approach can be quite inefficient. Although it is cheap to access a single row, the total cost is still high because there are huge number of rows need to be accessed for a given volume. Another drawback of *schema 3* is that due to the partitioning mechanism in Cassandra, multiple rows belonging to the same volume can be distributed amongst different nodes, which may incur further delay in processing. In our user requirement analysis, we found that in most cases clients request the whole volume content for processing and this is how *schema 2* comes. In *schema 2*, we encapsulate the whole volume within a single row but only using regular columns. In this manner regular columns can be accessed efficiently while still keeping all accesses to a volume within one row. Experimental results demonstrate that *schema 2* has superior performance over other two schemas.

5. EXPERIMENTAL RESULTS OF QUERIES

For public domain collections, the user is allowed to retrieve partial content or the whole volume to client side where further analysis or processing can be made. The typical use case or query is that the user specifies filtering conditions (normally against metadata) and then the data store retrieves all qualified volume records based on user specified volume IDs. We note that in practice HTRC leverages Solr [1] to index metadata and full text and to provision query interfaces to users. This is because Solr's index capability is much more powerful than Cassandra. However, in this paper we focus on evaluating Cassandra's performance, therefore in the following we define three query cases that are issued to Cassandra directly.

- *Retrieve Volume IDs based on Given Publish Date (query case 1)* — In this scenario, the user specifies a publish date or data range and Cassandra retrieves and returns the volume IDs of all volumes published within that range. Publish date is stored as metadata and in most cases it is only precise to year. Therefore we relax the constraint when matching the query, e.g., even though the user may specify an exact date, all volumes published within that year will be returned. After obtaining qualified volume IDs, the client may issue further requests for volume contents based on volume IDs.

- *Retrieve Volume IDs based on Given Genre (query case 2)* — Query case 2 is quite similar to query case 1 and both cases follow the pattern: "Given filtering conditions on metadata, retrieve all qualified volume IDs for possible future processing". Genre metadata can be missing for certain volumes, under such a circumstance, we can either accept or reject volumes without genre info. We adopt a conservative strategy which simply rejects the volumes without genre info.

- *Retrieve the Whole Volume Content based on Given Volume ID (query case 3)* — Below is how users make use of the HathiTrust corpus. The user specifies filtering conditions to retrieve desired volumes, however, the filtering conditions may not be exact at the first trial, open or unselective queries can bring back too many qualified records, especially when we have a large corpus. Therefore instead of trying to retrieve the contents of all qualified records, only identifies (i.e. volume IDs in our case) are returned. This strategy allows users to make a flexible choice on further processing. When the qualified set is large, user can either choose to only select a smaller subset or to conduct further analysis/processing on the whole but in a batch fashion (i.e. batch processing of multiple smaller worksets). The former approach is particularly useful for tentative analysis/statistics when only a rough sampling is needed. Meanwhile, we can expect a quick turnaround time when the size of the subset is small. Then users may choose to either further process the whole set when some interesting results are observed or discontinue otherwise. As mentioned easilier, another advantage of returning volume IDs only during filtering stage is that it allows the whole large workset to be decomposed into multiple smaller ones, which can be executed in batch or in parallel. Furthermore, this approach allows some intermediate results to be generated and returned earlier along with the processing without having to wait until the whole processing is done. In the above sense, case 3 can be viewed as a further query after cases 1 and 2.

To get reliable results, each user query is repeated 20 times. Figures 6(a) to 6(c) show the results of publish date (publish date set to be '2009'), genre (genre set to be 'Science & Technology') and whole volume content (volume id set to be 'iu.30000099847570') queries, respectively. X-axis shows the sequence number of each independent run and Y-axis shows the time consumed to fulfill the query. In all three queries we observe a decrease in response time after the first query request. The reason is that Cassandra caches the query results so that subsequent queries can be directly fulfilled by reading the cache without further processing. As we expect, *schema 2* performs the best in the three query cases.

6. EXPERIMENTAL RESULTS OF SIMULATED WORKLOADS

In this section we present experiments on IU collection that measure fine grained system level activities such as CPU, memory and IO usages as well as Cassandra' internal data structures like SSTable, MemTable and commit log, under five simulated workloads. We first define the simulated workloads and then detail the experimental results.

6.1 Workloads Definition

The following simulated workloads are defined for performance evaluation.

- *Workload 1* — We define workload 1 as the baseline case. We set up a moderate number of readers and writers, e.g., 10 readers and 5 writers. A writer reads first then writes, which is how writes are performed. Under this workload 75% of the requests are read requests and 25% are write requests. Each request touches a random 0.5% piece (140 volumes) of the data store, meaning each reader or writer reads or writes an average of 280 MB of data on a data store that is 56.4 GB (28,896 volumes) in size, not counting replication. Each reader/writer issues 20 requests before terminating. We ensured randomness through a random shuffle of the data set from which volumes are drawn; the client picks the first 0.5% piece from the data set.

- *Workload 2* — Workload 2 uses the same settings as workload 1 except that each client accesses the same set of volumes.

- *Workload 3* — Workload 3 uses the same settings as workload 1 except that the interval of two consecutive requests is 0.1 second instead of 1 second.

- *Workload 4* — Workload 4 uses the same settings as workload 3 except that each client accesses the same set of volumes.

- *Workload 5* — Workload 5 is more like the real scenario where the volumes IDs are obtained by first querying the Solr index. The reason why an extra index engine is introduced is that Cassandra's build in query language CQL and index are still quite limited and are not powerful enough to support various flexible queries to meet HTRC users' need. Since we are measuring the performance of Cassandra and do not want to include the cost of Solr query. Solr index is queried and the returned volume IDs are cached before clients start to make requests to Cassandra. Also note that there can be many qualified volumes given an open query, therefore the client may only pick a subset to further access Cassandra. To be more specific, workload 5 is composed of following four sorts of clients and each client only makes read request.

  Client type 1: There are 5 clients of this type. The query retrieves the IDs of the volumes whose author name contains keyword "edward". The query returns 399 volumes of which 156 are accessed.

  Client type 2: There are 5 clients of this type. The query retrieves the IDs of the volumes published between year 1970 and year 1979, both inclusive. The query returns 394 volumes of which 54 are accessed.

  Client type 3: There are 5 clients of this type. The query retrieves the IDs of the volumes whose title contains keyword "science". The query returns 535 volumes of which 246 are accessed.

Client type 4: There are 5 clients of this type. The query retrieves the IDs of the volumes whose title contains keyword "art". The query returns 222 volumes of which 215 are accessed.

6.2 Experimental Setup

We use the same 3-node Cassandra cluster as descried in Section 4 for workloads evaluation. To measure fine grained system level activities, on each Cassandra node we set up a *monitor* which measures following five metrics at a configurable interval (set to be 0.5 second in the experiments): CPU usage in percentage, memory usage in percentage, data read in megabytes per second, data written in megabytes per second and total IO (both read and write) usage in percentage. Note that since each node is a two-core machine, the total available CPU is 200%. The monitor is started 3 seconds before the trial and shut down 3 seconds after the trial. This buffering time allows us to observe the raise and drop of the measured metrics. Furthermore, we use a *sliding window* of size 5 to smooth the curve, the value of the data point $x(t)$ at time t showed in following figures is the averaged values in the window, i.e., $1/5*(x(t-2)+x(t-1)+x(t)+x(t+1)+x(t+2))$.

All clients run on a single powerful machine with 24 Intel (R) Xeon (R) cores of 2.00 GHz, and 126GB main memory (i.e. smoketree.cs.indiana.edu). The client machine and the Cassandra cluster sit on different subnets. Clients use hector API to make read/write requests to Cassandra with the default 'quorum' consistency level, the formula used to calculate the 'quorum' is (replication_factor / 2) + 1, which is 2 in our case, which means that all two replicas need to be read/written before returning success. Apart from aforementioned 5 system level metrics, we also measure response time, which is the length of time between submission of a job by a client, and the receipt of the last byte of the response.

6.3 Results

In this section we present extensive experimental results and give our discussion. We first detail the results on workload 1 and present our observations by dividing the total experimental time frame into four phases: warmup, steady state, flush to new SSTable and write only behavior. We then compare to workload 1 when discussing results of other workloads.

Warmup. Response time for workload 1, plotted in Figure 7(a), shows an initial substantial drop. The server upon startup reads the disk to load requested data from *SSTables* (the persistent form of data stored on disk) into in memory structures, *MemTable*. This is also reflected as a memory utilization burst in Figure 7(c) and disk read burst in Figure 7(d) over the same initial time frame. This cache warmup phase is dominated by intensive IO that is evidenced elsewhere as well. In Figure 7(e), we can see disk I/O activity that is higher during the warmup so CPU utilization is low during this time as shown in Figure 7(b).

Steady state. After the initial "warm-up" phase, we can see a relatively steady period in time frame [200 600] from Figure 7(a). As shown in Figures 7(b) and 7(c), the server maintains high CPU and memory utilizations in this period. Since each Cassandra node has moderate amount of memory, cache swaps are frequent which we see as persistent disk reads in Figure 7(d). In the meantime, we can also observe an increasing trend of disk writes in Figure 7(e). This is because apart from writing to the *commit log*, each write has also to be written into the in-memory *MemTable*.
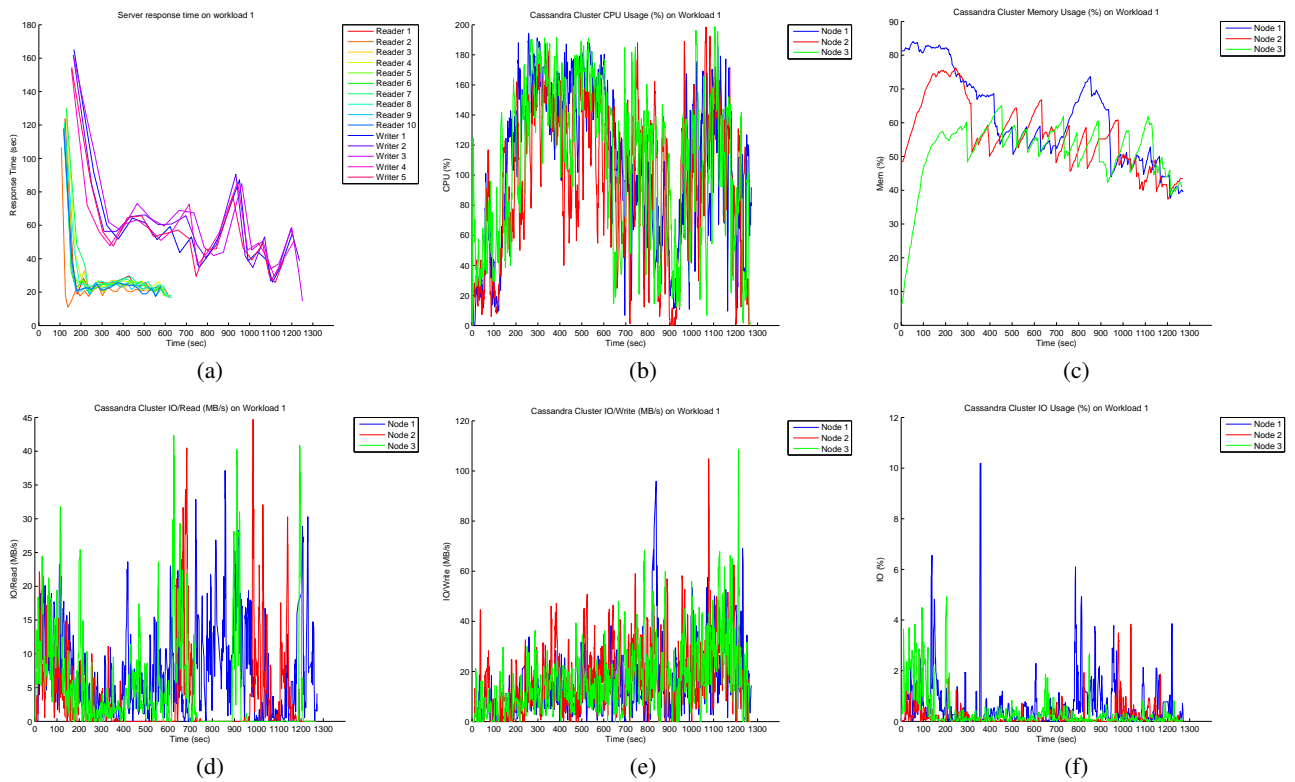
Figure 7: Performance measures of Cassandra cluster under workload 1. (a) server response time, (b) cluster CPU usage, (c) cluster memory usage, (d) cluster data read, (e) cluster data written, and (f) cluster IO usage.
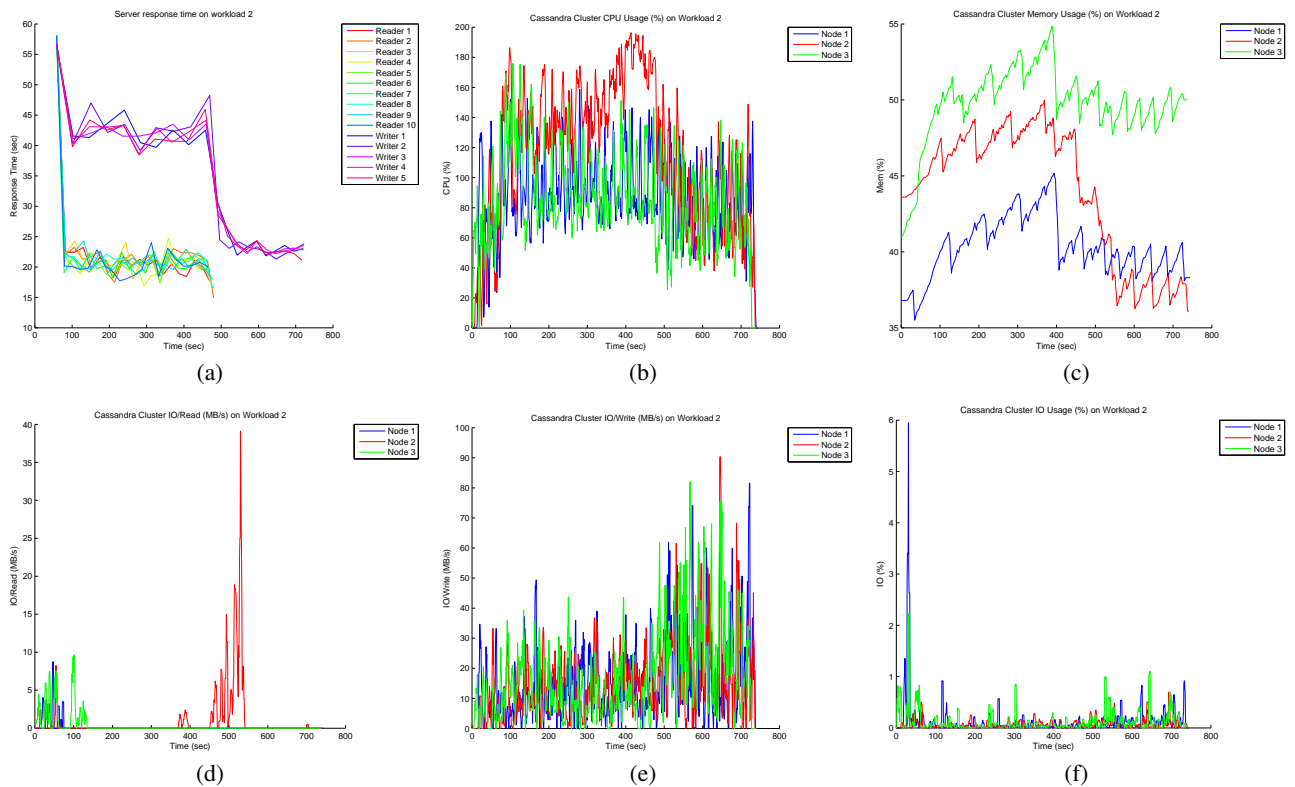

Figure 8: Performance measures of Cassandra cluster under workload 2. (a) server response time, (b) cluster CPU usage, (c) cluster memory usage, (d) cluster data read, (e) cluster data written, and (f) cluster IO usage.
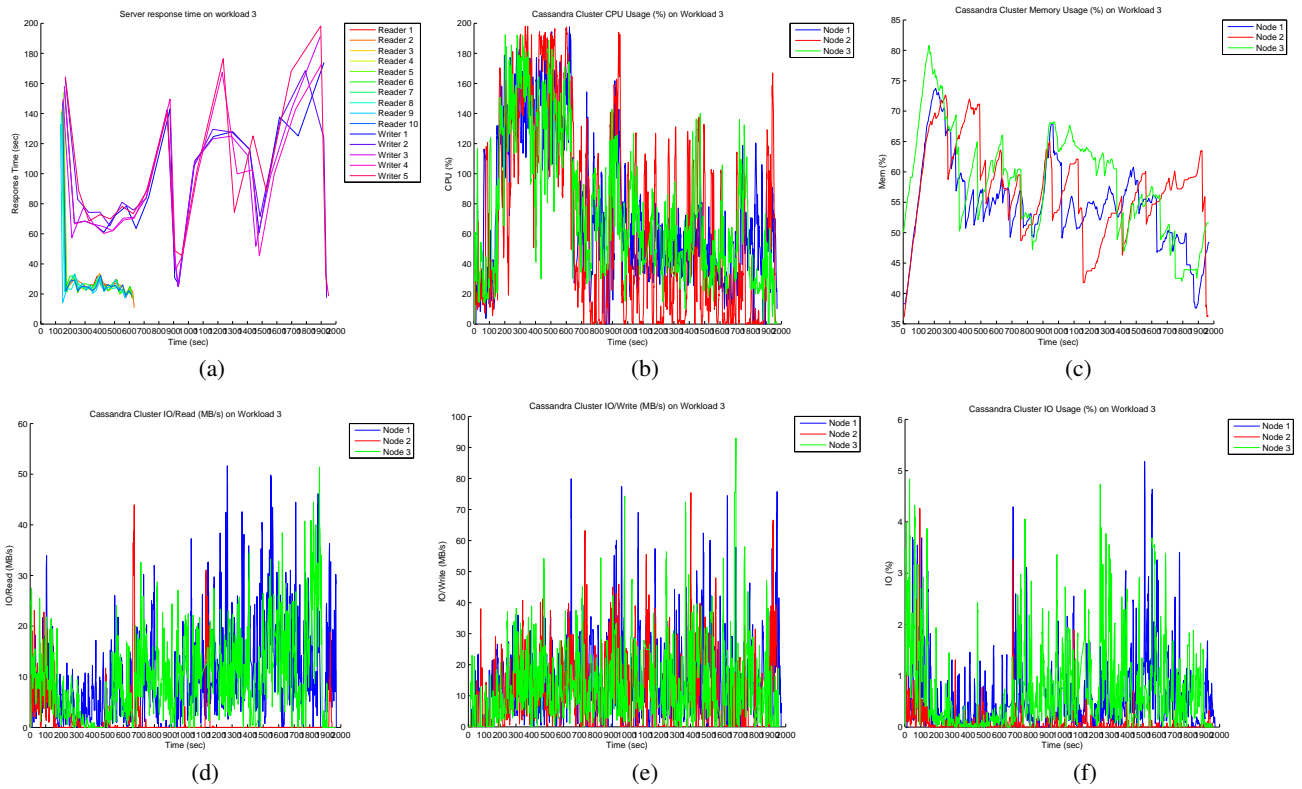
Figure 9: Performance measures of Cassandra cluster under workload 3. (a) server response time, (b) cluster CPU usage, (c) cluster memory usage, (d) cluster data read, (e) cluster data written, and (f) cluster IO usage.

Flush to new SSTable. As the size of *MemTable* reaches a threshold, they are flushed onto disk in the form of new *SSTables*. This activity is evidenced in Figure 7(a) during the time frame [600 800], where all readers finish their requests at around timestamp 650 and another drop in response time for writers, which benefits from readers' completion, since server can utilize all resources to serve writes since then.

Write only behavior. Once the readers complete, which happens at around the 700th second of execution, only the write workload remains. The absence of readers has the effect of freeing *MemTable* spaces that can then be used for writes (not all reads are halted since a writer also generates read requests), which would in turn generates more *MemTable* flushes. Meanwhile, the frequency of writes to *commit log* and to *MemTables* goes up since CPU resource are also dedicated to writers. At the same time, the *MemTable* spaces released by readers can allow new data to be in and thus generating further disk reads.

At 900 seconds into the experiment, there is a marked increase in response time (shown in Figure 7(a)) and drop in CPU utilization (shown in Figure 7(b)). I/O write is in a trough at that time too (shown in Figure 7(e)). But memory utilization for Node 1 is just coming off a big spike (shown in Figure 7(c)). It is difficult to know exactly what causes this behavior. It appears to be memory contention at Node 1 for the 5 writers that are remaining after the workers have finished.

Figure8(a) shows response time for workload 2. Different from workload 1, each client in workload 2 requests the same set of vol-umes. Under such a setting, Cassandra cluster is able to cache most read requests in *MemTable* after initial misses. Figure 8(d) shows that after the initial bursts the subsequent read IO activities are very low. For the write operation, different write requests modify the same portion of *Memtable*, which greatly reduces the number of disk writes caused by *Memtable* flush. However, writes to *commit log* are inevitable, hence we still observe continuous disk write activities in Figure 8(e). Also because *Memtable* almost always has available space, the cluster is able to acknowledge write success to clients immediately after writing the *commit log* and updating the *Memtable* without having to wait previous writes to be flushed first in order to get free spaces in *Memtable*, which is the case in workload 1. Therefore workload 2 has faster write response time than workload 1. The memory consumption of workload 2 is lower than workload 1 as shown in Figure 8(c), which benefits from being able to cache most read requests. As workload 2 requires less IOs, we also observe higher CPU usage than workload 1. Since read requests are faster than write requests, most readers finish earlier than writers and since then the load on the cluster is alleviated. Subsequent write requests thus can use more cluster resources, leading to a drop in the response time. The drop of writers' response times is observed in Figure 8(a) when most readers are done.

Workload 3 is similar to workload 1 but with more intensive request rate, which makes the competition of memory and disk IO on Cassandra cluster more intense. Compared with workload 1, we observe more intensive activities in IO (shown in Figures 9(d) and 9(e)) and more CPU idle times (shown in Figure 9(b)). The writers' response times (shown in Figure 9(a)) also have larger variation due to more intense resource competition.
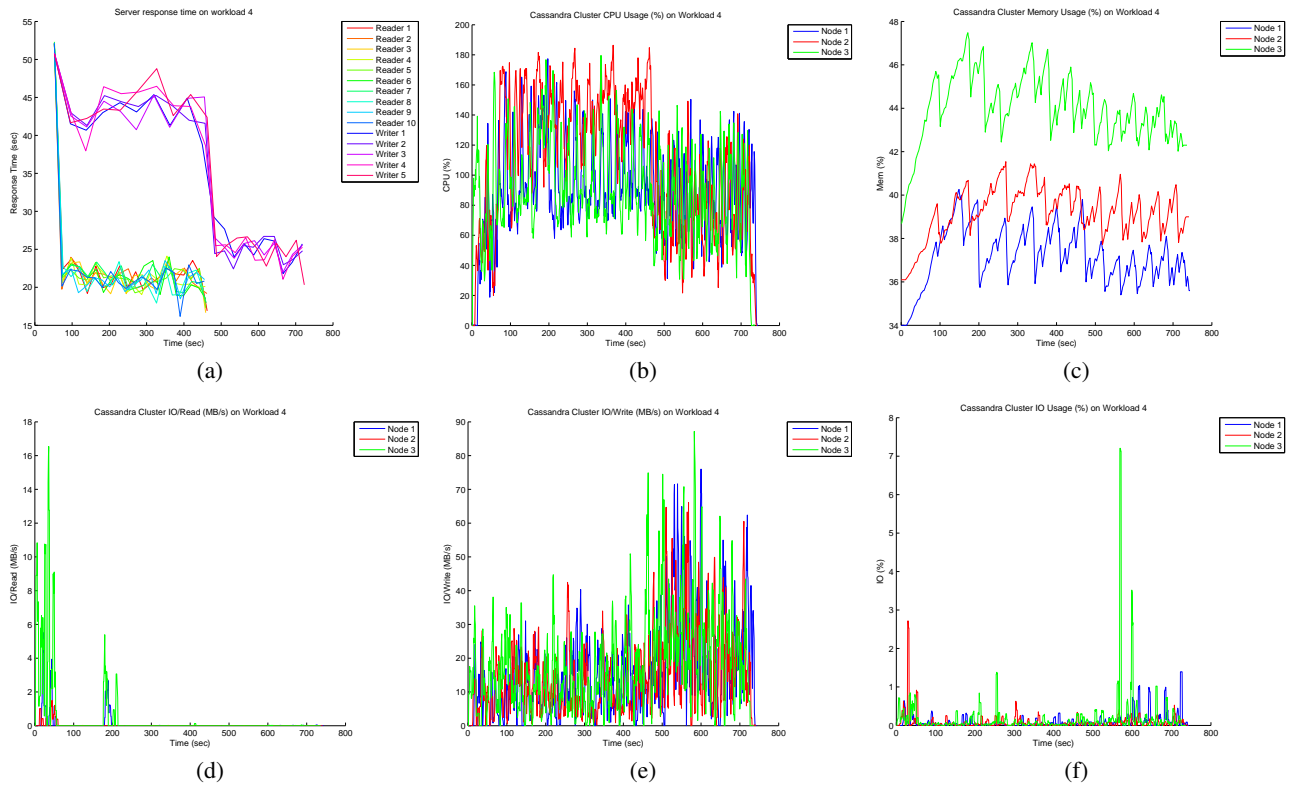
Figure 10: Performance measures of Cassandra cluster under workload 4. (a) server response time, (b) cluster CPU usage, (c) cluster memory usage, (d) cluster data read, (e) cluster data written, and (f) cluster IO usage.
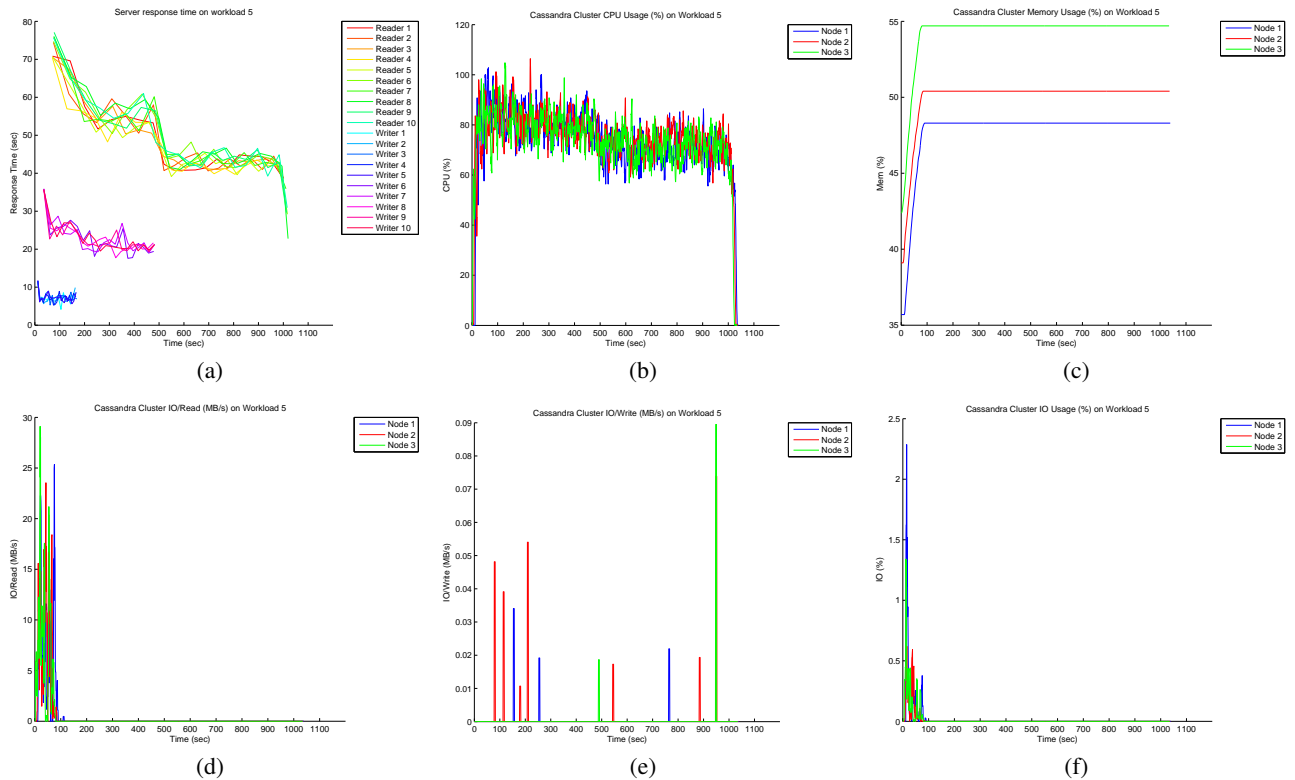

Figure 11: Performance measures of Cassandra cluster under workload 5. (a) server response time, (b) cluster CPU usage, (c) cluster memory usage, (d) cluster data read, (e) cluster data written, and (f) cluster IO usage.

Workload 4 is similar to workload 2 but with more intensive request rate. However, unlike workload 3 which suffers performance degradation due to the increased request rate, Cassandra cluster is able to handle the requests easily since the same set of requested volumes in workload 4 can still be cached. So the increased request rate does not have very obvious influence on workload 4. The results are shown from Figures 10(a) to 10(f).

Workload 5 is a relatively lightweight one and is only composed of readers. Under such settings, there are no writes to *commit log* and no need to flush *MemTables* to disk. Therefore there are only tiny disk write activities as shown in Figure 11(e). Moreover, after initial access to *SSTables* which loads data into *MemTables*, all subsequent read requests can be fulfilled without issuing further disk reads. So in Figure 11(d) we only observe IO reads bursts at the very beginning. We can also see relatively stable CPU usage curves from Figure 11(b) since no IO waits occur. Memory usage also maintains at a stable level as shown in Figure 11(c).

7. CONCLUSION

The relational model of data was proposed in 1970 by Ted Codd as the theoretical foundation for relational databases, which were quite successful in the past few decades. However, relational model is intended to be a useful approach of representing structure, applicable to certain problems, but not to be exhaustive. Relational applications encounter scalability problems when the workload goes up. Joins can be slow and the experience has shown that data stores that provide ACID guarantees tend to have poor availability. As performance and real time nature became more important than consistency for real-time web applications, alternatives need to be examined.

To address these issues, quite a few new NoSQL systems have been designed. These NoSQL systems are designed to be distributed, decentralized, elastically scalable, highly available, fault-tolerant, and tuneably consistent. These characteristics make NoSQL appealing to HTRC data store which needs to host large-scale repository of digital content and provision efficient read/write operations, as well as high scalability and availability. In this paper, we propose three column family schemas and perform extensive performance evaluations. Experimental results demonstrate that encapsulating the whole volume within a single row with regular columns delivers the best overall performance. Moreover, we perform evaluations with 5 simulated workloads and the experimental results allow us to examine behaviors of Cassandra's internal data structures like SSTables, Memtable and commit log, as well as system level activities such as CPU, memory and disk usages, which gives us a deeper understanding of Cassandra's working mechanisms.

8. REFERENCES

[1] Apache Solr. `https://lucene.apache.org/solr/`.
[2] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI'06)*, pages 335–350, Seattle, WA, USA, Nov 2006.
[3] R. Cattell. Scalable SQL and NoSQL data stores. In *ACM SIGMOD Record*, volume 39, pages 12–27, 2010.
[4] 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)*, 26(2), June 2008.
[5] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.
[6] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, Aug 2008.
[7] CouchDB. `http://couchdb.apache.org/`.
[8] 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 *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles (SOSP'07)*, pages 205–220, Stevenson, WA, USA, Oct 2007.
[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the 19th ACM symposium on Operating systems principles (SOSP'03)*, pages 29–43, Sagamore, NY, USA, Oct 2003.
[10] HathiTrust Digital Library. `http://www.hathitrust.org/`.
[11] HathiTrust Research Center. `http://www.hathitrust.org/htrc/`.
[12] HBase. `http://hbase.apache.org/`.
[13] HyperTable. `http://hypertable.org/`.
[14] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, Apr 2010.
[15] MongoDB. `http://mongodb.org/`.
[16] Pairtree. `https://confluence.ucop.edu/display/Curation/PairTree/`.
[17] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 3 edition, Aug. 2002.
[18] Riak. `http://basho.com/Riak.html/`.
[19] Scalaris. `http://code.google.com/p/scalaris/`.
[20] SimpleDB. `http://amazon.com/simpledb/`.
[21] Terrastore. `http://code.google.com/p/terrastore/`.
[22] Voldemort. `http://project-voldemort.com/`.