Investigation and Comparison of Distributed NoSQL Database Systems

Xiaoming Gao
Indiana University

This report investigates and compares four representative distributed NoSQL database systems, including HBase, Cassandra, MongoDB, and Riak, in terms of five dimensions: data model, data distribution mechanism, data replication and consistency management, data indexing support, and distributed data processing support.

1. Data Model

Data model defines the logical organization of data that is presented to the user or client application by a NoSQL database system.

1.1 HBase

HBase supports the BigTable data model [1] that was originally proposed by Google. Figure 1 illustrates this data model. Data are stored in **tables**; each table contains multiple **rows**, and a fixed number of **column families**. For each row, there can be a various number of **qualifiers (columns)** within each column family, and at the intersections of rows and qualifiers are table **cells**. Cell contents are uninterpreted byte arrays. Cell values are versioned using **timestamps**, and a table can be configured to maintain a certain number of versions. **Rows are sorted** by row keys, which are also implemented as byte arrays. Within each column family, columns are sorted by column names. Cell values under a column are further sorted by timestamps.

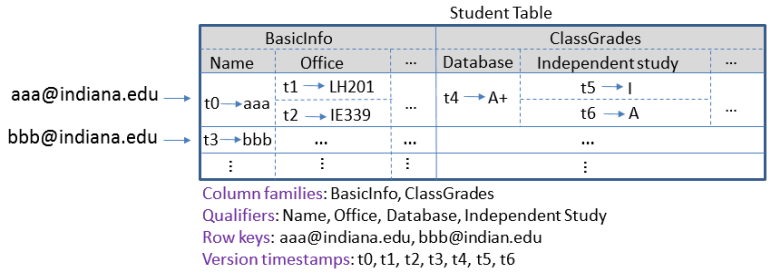

Figure 1. An example of the BigTable data model.

Compared with the data model defined by "relations" in traditional relational databases, HBase tables and columns are analogous to tables and columns in relational databases. However, there are **four significant differences**:

(1) Relational databases do not have the concept of "column families". In HBase, data from different columns under the same column family are stored together (as one file on HDFS). In comparison, data storage in relational databases is either row-oriented, where data in the same row are consecutively stored on physical disks, or column-oriented, where data in the same column are consecutively stored.

(2) In relational databases, each table must have a fixed number of columns (or "fields"). I.e. every row in a given table has the same set of columns. In HBase, each row in a table can have a different number of columns within the same column family.
(3) In HBase, cell values can be versioned with timestamps. The relational data model does not have the concept of versions.
(4) In general, NoSQL databases such as HBase do not enforce relationships between tables in the way relational databases do through mechanisms such as foreign keys. User applications have to deal with dependencies among tables through their application logics or mechanisms such as "Coprocessors" supported by HBase [5].

1.2 Cassandra

The data model of Cassandra [2][14] is overall similar to HBase, but with **several major differences**:

(1) In Cassandra, the concept of a **table** is equal to a "**column family**"; i.e. each table contains only one column family. Different column families are totally separate logical structures containing different set of row keys. Therefore, compared with the relational data model, Cassandra column families are analogous to tables, and **columns** under column families are analogous to columns in relational tables. Consider the example in Figure 1. In Cassandra, the "Student Table" in Figure 1 will be implemented either as one "Student" column family containing all the columns in Figure 1, or as two separate column families, "Student-BasicInfo" and "Student-ClassGrades".
(2) Beyond column families, Cassandra supports an extended concept of "**super column family**", which can contain "**super columns**". A super column is comprised of a (super) column name and an ordered map of **sub-columns**. The limitation of super columns is that all sub-columns of a super column must be deserialized in order to access a single sub-column value.
(3) The order of row keys in a column family depends on the data partition strategy used for a Cassandra cluster. By default the *Random Partitioner* is used, which means row keys are not sorted within a column family and there is no way to do range scans based on row keys without using external facilitating mechanisms such as an extra user-defined indexing column family. Row keys are sorted when the *Order Preserving Partitioner* is used, but this configuration is not recommended [3][4].
(4) Cassandra does not support explicit maintenance of multiple 'versions' of the **column (cell) values**. Column values do have associated timestamps but they are internally used for resolving conflicts caused by **eventual consistency**. Column values with obsolete timestamps are eventually deleted as a result of conflict resolution.

1.3 MongoDB

MongoDB is a distributed document database that provides high performance, high availability, and automatic scaling. It uses the concept of "**collections**" and "**documents**" to model data [6]. A collection is a grouping of MongoDB documents which normally have similar schemas. A collection is analogous to a table in relational databases and a document is analogous to a table record. Documents are modeled as a data structure following the JSON format, which is composed of field and value pairs. Each document is uniquely identified by a "_id" field as the primary key. The values of fields may include embedded documents, arrays, and arrays of documents [7]. Figure 2 shows an example MongoDB document. MongoDB can support access to a sorted list of documents by performing a query with sorting on a document field [].

Figure 2. An example of the MongoDB document data model [7].

Relationships between documents can be modelled in two ways: references and embedded documents [8].

1.4 Riak

Riak is a distributed database designed for key-value storage. Its data model follows a simple "**key/value**" scheme, where the key is a unique identifier of a data object, and the value is a piece of data that can be of various types, such as text and binary [10]. Each data object can also be tagged with additional metadata, which can be used to build secondary indexes to support query of data objects [11]. A concept of "**bucket**" is used as a namespace for grouping key/value pairs. Figure 3 illustrates an example of the Riak data model.

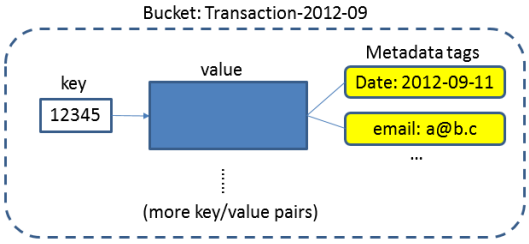

Figure 3. An example of the key/value data model in Riak.

2. Data Distribution Mechanism

The data distribution mechanism determines how data operations are distributed among different nodes in a NoSQL database cluster. Most systems use two major mechanisms: **key-range based** distribution and **hash based** distribution. Key-range based distribution can easily support range scans of sorted data, but may face the problem of unbalanced access load to different value ranges. Hash based distribution has the advantage of balanced access load across nodes, but does not support range scans very well.

2.1 HBase

HBase uses a **key-range based** data distribution mechanism. Each table horizontally split into regions, and regions are assigned to different region servers by the HBase master. Since rows are sorted by row keys in the HBase data model, each region covers a consecutive range of row keys. Figure 4 illustrates the architecture of HBase. HBase dynamically splits a region into two when its size gets over a limit, or according to a user-specified ***RegionSplitPolicy***. Users can also force region splits to handle "hot" regions [11]. Since table data are stored in HDFS, region splits do not involve much data movement and can be finished very quickly. Region splits happens in the background and does not affect client applications.

2.2 Cassandra

Depending on the configuration about *data partitioner*, a Cassandra cluster may apply either **key-range based** distribution or **hash based** distribution.

When the *Random Partitioner* is used (which is the default configuration), nodes in the cluster form a Distributed Hash Table (DHT). Cassandra partitions data across the cluster using consistent hashing. The output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its position on the ring. After position assignment, each node becomes responsible for the region in the ring between it and its predecessor node on the ring [14].

To handle a data operation request, the row key of the data operation is first hashed using the MD5 hashing algorithm, and then the operation is sent to the node that is responsible for the corresponding hash value to process. The MD5 hashing step ensures a balanced distribution of data and workload even in cases where the application data has an uneven distribution across the row keys, because the hash values of the possibly preponderant sections of row keys will still demonstrate an even distribution [4].

When the *Order Preserving Partitioner* is used, each node becomes responsible for the storage and operations of a consecutive range of row keys. In this case, when the application data has an uneven distribution across the row key space, the nodes will have an unbalanced workload distribution [4].

Load skew may be further caused by two other factors. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic data distribution algorithm is oblivious to the heterogeneity in the performance of nodes. To address these issues, Cassandra analyzes load information on the ring and **move lightly loaded nodes on the ring** to alleviate heavily loaded nodes [14]. Besides, every time a new node is added, Cassandra will assign a range of keys to that node such that it takes responsibility for half the keys stored on the node that currently stores the most keys. In a stable cluster, data load can also be rebalanced by careful administrative operations, such as manual assignment of key ranges or node take-down and bring-up [4].

2.3 MongoDB

MongoDB also supports both **key-range based** distribution and **hash based** distribution through configurations. The working logic is similar to Cassandra. MongoDB organizes nodes in units of **shards** and partitions the key space of data collections into **chunks**. Chunks are then distributed across the shards. Dynamic load balancing among shards are achieved through ***chunk splitting*** and ***chunk migration*** [13].

2.4 Riak

Riak also uses a DHT to support **hash based** distribution. When the client performs key/value operations, the bucket and key combination is hashed. The resulting hash maps onto a 160-bit integer space. Riak divides the integer space into equally-sized partitions. Each partition is managed by a process called a virtual node (or "vnode"). Physical machines evenly divide responsibility for vnodes. Figure 4 [10] illustrates an example partition distribution of the hash value space among 4 nodes.

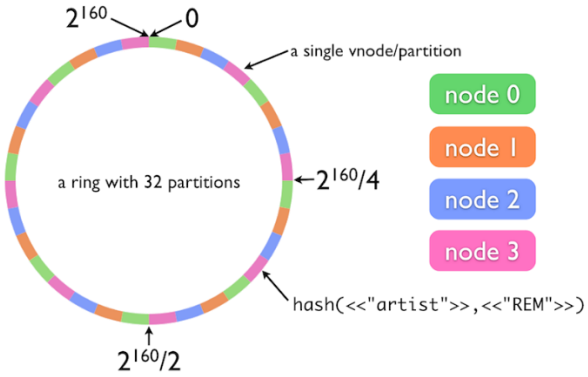

Figure 4. Hash based data distribution in Riak [10].

3. Data Replication and Consistency Management

Almost all NoSQL database systems rely on replication to ensure high data availability in distributed deployments. However, different systems use different strategies to manage the consistency of multiple replicas of the same piece of data. This section only covers **data-object-level** consistency, i.e. consistency among replicas of single data objects. Most NoSQL database systems do not address **transaction-level** consistency, which may involve a series of updates to multiple related data objects. Supporting transaction-level consistency will require additional synchronization extensions [13].

3.1 HBase

Since HBase uses HDFS for data storage, it inherits the **replication and consistency management from HDFS**. Specifically, the replication factor and replica location method is decided by HDFS. Since HDFS enforces complete consistency – a write operations does not return until all replicas have been updated – HBase also ensures **complete consistency** for its data update operations. Upon receiving a data update operation, the HBase region server first records this operation in a write-ahead log (WAL), and then put it in its *memstore* (an in-memory data structure). When the *memstore* reaches its size limit, it is written to an HFile [15]. Both the WAL file and the store file are HDFS files. Therefore, complete consistency is guaranteed for all data updates. HDFS and HBase do not originally support deployment with **data center awareness**.

3.2 Cassandra

Each data item in Cassandra is replicated at N hosts, where N is the replication factor. The node responsible for the key of the data item is called a coordinator node. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 nodes in the ring. Cassandra provides various **replication policies** such as "**Rack Unaware**", "**Rack Aware**" (within a datacenter) and "**Datacenter Aware**". Replicas are chosen based on the replication policy chosen by the application. If the "Rack Unaware" replication strategy is chosen, then the non-coordinator replicas are chosen by picking N-1 successors of the coordinator on the ring.

Cassandra allows **eventual consistency** among data replicas to achieve high availability, partition tolerance and short response time for data operations. Cassandra extends the concept of eventual consistency by offering **tunable consistency**. For any given read or write operation, the client application decides how consistent the requested data should be. The consistency level can be

specified using values such as “ANY”, “ONE”, “QUORUM”, “ALL”, etc. Some values are specially designed for multiple data center clusters, such as “LOCAL_QUORUM” and “EACH_QUORUM” [16]. To understand the meaning of consistency levels, take “QUORUM” for write as an example. This level requires that a write operation will be sent to all replica nodes, and will only return after it is written to the commit log and memory table on a quorum of replica nodes.

Cassandra provides a number of built-in repair features to ensure that data remains consistent across replicas, including ***Read Repair***, ***Anti-Entropy Node Repair***, and ***Hinted Handoff*** [16].

3.3 MongoDB

MongoDB manages data replication in the units of **shards**. Each shard is a replica set, which can contain one **primary** member, multiple **secondary** members, and one **arbiter**. The primary is the only member in the replica set that receives write operations. MongoDB applies write operations on the primary and then records the operations on the primary’s oplog. Secondary members replicate this log and apply the operations to their data sets. All members of the replica set can accept read operations. However, by default, an application directs its read operations to the primary member. If the current primary becomes unavailable, an election determines the new primary. Replica sets with an even number of members may have an arbiter to add a vote in elections of for primary [17]. Replica sets can be made **data center-aware** through proper configurations [18].

Data synchronization between primary and secondaries are completed through **eventual consistency** [19]. If *Read Preference* is set to **non-primary**, read operations directed to secondaries may get stale data [20]. MongoDB also supports **tunable consistency** for each write operation through the “Write Concern” parameter []

3.4 Riak

Riak allows the user to set a **replication number** for each **bucket**, which defaults to 3. When a data object's key is mapped onto a given partition of the circular hash value space, Riak automatically replicates the data onto the next two partitions (Figure 5) [10]. Riak supports multi data center replication through the concept of “primary cluster” and “secondary clusters” [22].

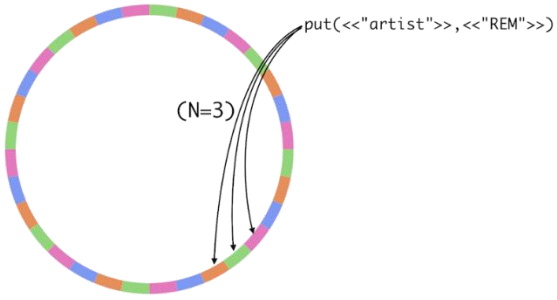

Figure 5. Data replication in Riak [10].

Similar to Cassandra, Riak also supports **tunable consistency** for each data operation [21]. It relies on mechanisms such as ***Vector Clock***, ***Hinted Handoff***, and ***Read Repair*** to resolve conflicts and ensure consistency [10].

4. Data Indexing Support

There are two major categories of indexing involved in distributed NoSQL database systems: **primary indexing** and **secondary indexing**. In terms of distributed index storage, there are two ways of **index partitioning**: **partition by original data** or **partition by index key**. "Partition by original data" means that each node in the cluster only maintains the secondary index for the portion of the original data that is locally hosted by this node. In this case, when a query involving an indexed field is evaluated, the query must be sent to every node in the cluster. Each node will use the local portion of secondary index to do a "partial evaluation" of the query, and return a subset of result. The final result is generated by combining results from all the nodes. Figure 6 illustrates partition by original data. "Partition by index key" means that a global index is built for the whole data set on all the nodes, and then distributed among the nodes by making partitions with the key of the index. To evaluate a query about an indexed field value, only the node maintaining the index for that queried field value is contacted, and it processes all related index entries to get the query result. Figure 7 illustrates partition by index key.

Figure 6. Partition by original data.

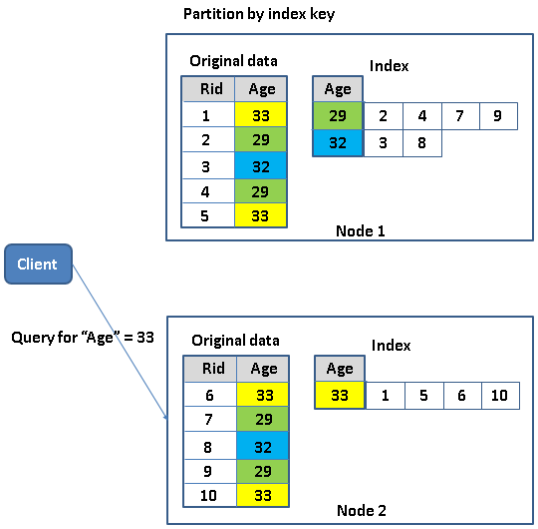

Figure 7. Partition by index key.

Partition by original data is good for handling complicated queries involving multiple fields and constraints, because each node can partially evaluate the query by only accessing local data. Although the query has to be broadcast to all nodes, the total amount of communication is much smaller than the size of the relevant part of the indexes for each field. Partition by index key works better when queries are simple, because the major part of evaluation is the processing and transmission of the related index entries, and only the exact related node(s) need to be contacted.

4.1 HBase

Primary indexing

HBase builds a **primary index** on the row keys, which is conceptually similar to a distributed multi-level **B+-tree** index. HBase maintains two global catalog tables: ***ROOT*** and ***META***. *ROOT* always has only one region, and its location is stored in ***ZooKeeper***. *ROOT* keeps track of the regions of the *META* table, and *META* keeps a list of all regions in the system, as well as which region servers are hosting them [24]. On the region server, data are read from and written to HFiles on HDFS, and the HFile format contains information about a multi-level B+-tree like data

structure [23]. The primary index is a clustered index because the data records are stored directly in the index entries.

Secondary Indexing

HBase does not originally support secondary indexes for cell values. IndexedHBase extends HBase with the capability of defining and building customizable index structures using HBase tables. By using the index configuration file and user-defined pluggable indexer implementation, it is possible to define various index structures, including **B+-tree-like single field index**, **multidimensional index**, **text index** [44], **geospatial index** [45], or even **multi-key index** as supported by MongoDB [30]. IndexedHBase has been successfully applied in building multidimensional text indexes for supporting social data analysis applications [25].

Consistency between data and index

Since IndexedHBase directly uses HBase tables to store indexes, index updates are not atomically associated with data updates. Eventual consistency between index updates and data updates is completed at the level of milliseconds.

Secondary index partition scheme

Since cell values of the data table are used as row keys for the index tables, the index tables are **partitioned by index keys**. A hybrid solution of partition by original data and partition by index key can be achieved by first partition the original data table into multiple "sub-tables", and then build an index table for each "sub-table". For example, in our experience in supporting social data analysis applications [25], original data are partitioned by month into different tables, and then separately indexed with index tables.

4.2 Cassandra

Primary indexing

The DHT architecture of Cassandra basically builds a distributed **primary key hash index** for the row keys of column families. This primary index is a **clustered index** since data records are contained in the index entries.

Secondary Indexing

Beyond primary key index, Cassandra supports creation of secondary indexes on any column values [26]. The internal secondary index implementation depends whether the data type of the column values is **non-text** data and **text** data.

For **non-text** column values, Cassandra can create **hash indexes** which are internally maintained as **hidden index column families** [27]. This index column family stores a mapping from index values to a sorted list of matching row keys. Since the index is a hash index, query results are not sorted by the order of the indexed values. Besides, range queries on indexed columns cannot be completed by using the index. Although an "equal" match in the index returns an ordered list relevant row keys.

For **text** column values, the commercial version of Cassandar, DataStax, supports secondary indexes on text data through integration with Solr [28]. Moreover, the indexes are stored as

Lucene index files [29], which means various query types, including equal queries, wildcard queries, range queries, etc. can be supported.

Consistency between data and index

Data update + index update is an atomic operation, so **immediate consistency** is ensured between the original data and index data.

Secondary index partition scheme

Each node maintains the secondary indexes for its own local part of original data. Therefore, secondary indexes are **partitioned by original data**.

Limitations

Cassandra secondary indexes currently have several **limitations**. First, they can only index values from single columns; multidimensional indexes as used in [25] are not supported. Second, as mentioned above, indexes for **non-text** columns cannot be used to evaluate range queries. Finally, even if a query specifies constraints on multiple indexed columns, only one index will be used to quickly locate the related row keys. Range constraints can be specified on additional columns in the query, but are checked against the original data instead of using indexes [26].

4.3 MongoDB

Primary indexing

MongoDB automatically forces the creation of a **primary key index** on the *_id* field of the documents. Index entries are sorted by *_id*, but note that this primary key index is **not a clustered index** in Database terms. I.e. the index entries only contains pointers to actual documents in the MongoDB data files. Documents are not physically stored in the order of *_id* on disks.

Secondary Indexing

Beyond the primary index, MongoDB supports various secondary indexes for field values of documents, including single field index, multidimensional index, multikey index, geospatial index, text index, and hashed index [30]. Single field, multidimensional, and multikey indexes are organized using **B-tree** structures. The geospatial index supports indexing using **quad trees** [47] on 2-dimension geospatial data. The official documentation does not provide details about how the text indexes are implemented, but it is known that basic features such as stopping, stemming, and scoring are supported [48]. Text index in MongoDB is still in beta version. The hashed index can be used to support both hash based data distribution and equality queries of field values in documents, but obviously cannot be used for range queries.

Consistency between data and index

Data is indexed on the fly in the same atomic operation. Therefore, **immediate consistency** is ensured between the original data and index data.

Secondary index partition scheme

Each shard maintains the secondary index for its local partition of the original data. Therefore, secondary indexes are **partitioned by original data**.

4.4 Riak

Primary indexing

As explained in section 2.4, Riak builds a **primary key hash index** for its key/value pairs through DHT. This index is a **clustered index** because data objects are directly stored together with the index keys.

Secondary Indexing

Riak supports secondary indexes on the tagged attributes of the key/value pairs and inverted indexes for text data contained in the value. For secondary indexes on tagged attributes, exact match and range queries are supported. However, current Riak implementation forces the limitation that one query can only use secondary index search on one indexed attribute (field). Queries involving multiple indexed attributes have to be broken down as multiple queries; then the results are merged to get the final result [31]. No details are given about the internal structures used for secondary indexes in the official Riak documentation. According to the brief mention in [46], it seems that a flat list of key/entries is used.

For inverted indexes on values of text type, text data contained in the values of key/value pairs are parsed and indexed according to a pre-defined index schema. Similar to DataStax, Riak also tries to integrate with the interface of Solr, and stores index using the Lucene file format, so as to support various types of queries on text data, such as wildcard queries and range queries [32]. Beyond the basic inverted index structure, Riak supports a special feature called "inline fields" [33]. If a field is specified as an "inline" field, its value will be attached to the document IDs in the posting lists of every indexed field of the inverted index. Inline fields are useful for evaluating queries involving multiple fields of text data, but it is not flexible enough in the sense that the value of every inline field will appear in the posting list of every indexed field, which may cause unnecessary indexing and storage overhead.

Consistency between data and index

Data update + index update is an atomic operation, so **immediate consistency** is ensured between the original data and index data.

Secondary index partition scheme

For secondary indexes on tagged attributes, each node maintains the indexes for its local part of original data. Therefore, the indexes are **partitioned by original data**. However, the text index is **partitioned by terms** (keys in inverted index). In Riak, text index schemas are configured at the level of buckets. I.e. all the key/value pairs in a configured bucket will be parsed and indexed according to the same given schema. A global inverted index is created and maintained for all key/value pairs added to that bucket, and then partitioned by terms in the inverted index, and distributed among all the nodes in the ring.

5. Distributed Data Processing Support

5.1 HBase and Cassandra

HBase and Cassandra both support parallel data processing by integration with Hadoop MapReduce [34][35][36], which is designed for fault tolerant parallel processing of large batches

of data. It implements the full semantics of the MapReduce computing model and applies a comprehensive initialization process for setting up the runtime environment on the worker nodes. Hadoop MapReduce uses disks on worker nodes to save intermediate data and does grouping and sorting before passing them to reducers. A job can be configured to use zero or multiple reducers.

5.2 MongoDB

MongoDB provides two frameworks to apply parallel processing to large document collections: **aggregation pipeline** [37] and **MapReduce** [38].

The aggregation pipeline completes aggregate computation on a collection of documents by applying a pipeline of data operators, such as *match*, *project*, *group*, etc. By using proper operators such as *match* and *skip* at the beginning of the pipeline, the framework is able to take advantage of existing **indexes** to limit the scope of processing to only a related subset of documents in the collection and thus achieve better performance. Currently MongoDB implementation enforces several important limits on the usage of aggregation pipelines, including input data types, final result size, and memory usage by operators [39]. This implies that the pipeline operators operate completely in memory and does not use external disk storage for computations such as sorting and grouping.

The MapReduce framework is designed to support aggregate computations that go beyond the limits of the aggregation pipeline, as well as extended data processing that cannot be finished by the aggregation pipeline. MapReduce functions are written in JavaScript, and executed in MongoDB daemon processes. Compared with Hadoop MapReduce, MongoDB MapReduce is different in several aspects. First, *reduce* is only applied to the *map* outputs where a key has multiple associated values. Keys associated with single values are not processed by *reduce*. Second, besides *map* and *reduce*, an extra *finalize* phase can be applied to further process the outputs from *reduce*. Third, a special "incremental MapReduce" mechanism is provided to support dynamically growing collections of documents. This mechanism allows *reduce* to be used for merging the results from the latest MapReduce job and previous MapReduce jobs. Fourth, the framework supports an option for choosing the way intermediate data are stored and transmitted. The default mode stores intermediated data on local disks of the nodes, but the client can specify to only use memory for intermediated data storage, in which case a limit is enforced on the total size of key/value pairs from the *map* output. Finally, functions written in JavaScript may limit the capabilities of *map* and *reduce*. For example, it is hard or even impossible to access an outside data resource such as a database or distributed file system [40][41] to facilitate the computation carried out in *map* and *reduce*.

5.3 Riak

Riak provides a lightweight MapReduce framework for users to query the data by defining MapReduce functions in JavaScript or Erlang []. Furthermore, Riak supports MapReduce over the search results by using secondary indexes or text indexes. Riak MapReduce is different from Hadoop MapReduce in several ways. There is always only one reducer running for each MapReduce job. Intermediate data are transmitted directly from mappers to the reducer without being sorted or grouped. The reducer relies on its memory stack to store the whole list of intermediate data, and has a default timeout of only five seconds. Therefore, Riak MapReduce is not suitable for processing large datasets.

6. Summary

Table 1 provides a summary of this report.

Table 1. Comparison of representative distributed NoSQL databases

	HBase	Cassandra	MongoDB	Riak
Data model	Table -> column family -> column -> version -> cell value	Table (column family) -> column -> cell value	Collection -> document	Bucket -> key/value with tagged metadata
Architecture	HMaster + region servers. HMaster handles failure recovery and region splits.	Decentralized DHT. Replication, failure recovery and node join/departure are handled in a decentralized manner.	Config servers + data operation routers + shards. Within a shard: one primary + multiple secondaries + arbiters. Replication and failure recovery handled at the shard level.	Decentralized DHT. Replication, failure recovery and node join/departure are handled in a decentralized manner.
Data distribution mechanism	Key-range based distribution.	Key-range based distribution and hash based distribution.	Key-range based distribution and hash based distribution.	Hash based distribution.
Data replication and consistency model	Decided by HDFS. No "data center aware" replica placement. Complete consistency.	Configurable replication policy with data center awareness. Tunable eventual consistency for each data operation.	Replication level configured through secondaries. Eventual consistency between primary and secondaries. Tunable eventual consistency for each operation.	Replication level configured at the bucket level. Tunable eventual consistency for each data operation.
Data indexing support	No original secondary indexing support. Customizable indexing with IndexedHBase. Eventual consistency between data and index.	Secondary hash indexes implemented as hidden column families for non-text data, and as Lucene files for text data. Nodes maintain secondary indexes locally. Atomic data + index updates.	B-tree for single-field, multidimensional, and multikey indexes. Quad tree for geospatial index. Text index, and hash index also supported. Indexes are locally maintained by shard members.	Secondary indexes for tagged metadata maintained locally, and text index for text data maintained globally. Atomic data + index updates.
Distributed data processing support	Hadoop MapReduce.	Hadoop MapReduce.	Aggregation pipeline for aggregation operations and Map-Reduce-Finalize for more complicated operations. Intermediate data storage can be configured to use disks or not.	A lightweight MapReduce framework for query purposes. In-memory intermediate data storage + single reducer.

References

[1] Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A. and Gruber, R. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating System Design and Implementation* (Seattle, WA, USA, November 06-08, 2006). OSDI '06. USENIX Association Berkeley, CA, 205-218.

[2] Understanding the Cassandra Data Model. *Apache Cassandra 0.8 documentation.* Available at http://www.datastax.com/docs/0.8/ddl/index.

[3] Partitioners. *Cassandra Wiki.* Available at http://wiki.apache.org/cassandra/Partitioners.

[4] Williams, D. 2010. Cassandra: RandomPartitioner vs OrderPreservingPartitioner. *Blog post* available at http://ria101.wordpress.com/2010/02/22/cassandra-randompartitioner-vs-orderpreservingpartitioner/.

[5] Lai, M., Koontz, E., Purtell, A. 2012. Coprocessor Introduction. *Apache HBase blog post* available at http://blogs.apache.org/hbase/entry/coprocessor_introduction.

[6] MongoDB Glossary. *MongoDB documentation* available at http://docs.mongodb.org/manual/reference/glossary/#term-collection.

[7] Introduction to MongoDB. *MongoDB documentation* available at http://docs.mongodb.org/manual/core/introduction/.

[8] Data Modeling Introduction. *MongoDB documentation* available at http://docs.mongodb.org/manual/core/data-modeling-introduction/.

[9] Reference for the "orderby" operator. *MongoDB documentation* available at http://docs.mongodb.org/manual/reference/operator/meta/orderby/.

[10] Riak introduction. *Riak documentation* available at http://docs.basho.com/riak/latest/theory/why-riak/.

[11] Soztutar, E. Apache HBase Region Splitting and Merging. *Blog post* available at http://hortonworks.com/blog/apache-hbase-region-splitting-and-merging/.

[12] Sharding Introduction. *MongoDB documentation* available at http://docs.mongodb.org/manual/core/sharding-introduction/.

[13] Peng, D., Dabek, F. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, (USENIX 2010).*

[14] Lakshman, A., Malik, P. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*. 44(2): 35-40. 2010

[15] George, L. HBase: The Definitive Guide. 2011. *O'Reilly Media, Inc*. September 2011.

[16] About Data Consistency in Cassandra. *Apache Cassandra 1.1 documentation.* Available at http://www.datastax.com/docs/1.1/dml/data_consistency.

[17] Replica Set Members. *MongoDB documentation* available at http://docs.mongodb.org/manual/core/replica-set-members/.

[18] Data Center Awareness. *MongoDB documentation* available at http://docs.mongodb.org/manual/data-center-awareness/.

[19] On Distributed Consistency - Part 2 - Some Eventual Consistency Forms. 2010. *MongoDB blog* available at http://blog.mongodb.org/post/498145601/on-distributed-consistency-part-2-some-eventual.

[20] Read Preference. *MongoDB documentation* available at http://docs.mongodb.org/manual/core/read-preference/.

[21] Eventual Consistency. *Riak documentation* available at http://docs.basho.com/riak/latest/theory/concepts/Eventual-Consistency/.

[22] Multi Data Center Replication: Architecture. *Riak documentation* available at http://docs.basho.com/riakee/latest/cookbooks/Multi-Data-Center-Replication-Architecture/.

[23] Bertozzi, M. Apache HBase I/O – HFile. 2012. Blog post available at http://blog.cloudera.com/blog/2012/06/hbase-io-hfile-input-output/.

[24] Catalog Tables. *HBase documentation* available at http://hbase.apache.org/book/arch.catalog.html.

[25] Gao X., Roth, E., McKelvey, K., Davis, C., Younge, A., Ferrara, E., Menczer, F., Qiu, J. 2013. Supporting a Social Media Observatory with Customizable Index Structures - Architecture and Performance. Book chapter to appear in *Cloud Computing for Data Intensive Applications*, to be published by Springer Publisher, 2014.

[26] About Indexes in Cassandra. *Cassandra 1.1 documentation* available at http://www.datastax.com/docs/1.1/ddl/indexes.

[27] How do secondary indices work? From the *Cassandra users mailing group*, available at http://cassandra-user-incubator-apache-org.3065146.n2.nabble.com/Re-How-do-secondary-indices-work-td6005345.html.

[28] DataStax Enterprise: Cassandra with Solr Integration Details. *DataStax Enterprise 2.0 documentation* available at http://www.datastax.com/dev/blog/datastax-enterprise-cassandra-with-solr-integration-details.

[29] Apache Lucene - Index File Formats. *Lucene documentation* available at http://lucene.apache.org/core/3_5_0/fileformats.html.

[30] Index Introduction. *MongoDB documentation* available at http://docs.mongodb.org/manual/core/indexes-introduction/.

[31] Using Secondary Indexes. *Riak documentation* available at http://docs.basho.com/riak/latest/dev/using/2i/.

[32] Using Search. *Riak documentation* available at http://docs.basho.com/riak/latest/dev/using/search/.

[33] Zezeski, R. Boosting Riak Search Query Performance with Inline Fields. 2011. *Blog post* available at http://basho.com/boosting-riak-search-query-performance-with-inline-fields/.

[34] MapReduce Tutorial. *Hadoop documentation* available at https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

[35] HBase and MapReduce. *HBase documentation* available at http://hbase.apache.org/book/mapreduce.html.

[36] Hadoop Support. *Cassandra wiki page* available at http://wiki.apache.org/cassandra/HadoopSupport.

[37] Aggregation Pipeline. *MongoDB documentation* available at http://docs.mongodb.org/manual/core/aggregation-pipeline/.

[38] Map-Reduce. *MongoDB documentation* available at http://docs.mongodb.org/manual/core/map-reduce/#map-reduce-behavior.

[39] Aggregation Pipeline Limits. *MongoDB documentation* available at http://docs.mongodb.org/manual/core/aggregation-pipeline-limits/.

[40] Limitations with JavaScript. From *online tutorial for JavaScript*, available at http://cbtsam.com/jsl1/cbtsam-jsl1-012.php.

[41] Chapman, S. What Javascript Can Not Do. *Online article* available at http://javascript.about.com/od/reference/a/cannot.htm.

[42] Using MapReduce. *Riak documentation* available at http://docs.basho.com/riak/latest/dev/using/mapreduce/.

[43] Write Concern. *MongoDB documentation* available at http://docs.mongodb.org/manual/core/write-concern/.

[44] Gao, X., Nachankar, V., Qiu. J. 2011. Experimenting Lucene Index on HBase in an HPC Environment. 2011. In *Proceedings of the 1st workshop on High-Performance Computing meets Databases at Supercomputing 2011*. Seattle, WA, USA, November 18, 2011.

[45] Nishimura, S., Das, S., Agrawal, D., Abbadi, A. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *Proceedings of the 2011 IEEE 12th International Conference on Mobile Data Management (MDM 2011)*. Luleå, Sweden, June 6-9, 2011.

[46] Advanced Secondary Indexes. Riak documentation available at http://docs.basho.com/riak/latest/dev/advanced/2i/.

[47] 2d Index Internals. MongoDB documentation available at http://docs.mongodb.org/manual/core/geospatial-indexes/.

[48] Text Indexes. MongoDB documentation available at http://docs.mongodb.org/manual/core/index-text/.