COMPARATIVE STUDY OF NOSQL DOCUMENT, COLUMN STORE DATABASES AND EVALUATION OF CASSANDRA

Manoj V

VIT University, India

ABSTRACT

In the last decade, rapid growth in mobile applications, web technologies, social media generating unstructured data has led to the advent of various nosql data stores. Demands of web scale are in increasing trend everyday and nosql databases are evolving to meet up with stern big data requirements. The purpose of this paper is to explore nosql technologies and present a comparative study of document and column store nosql databases such as cassandra, MongoDB and Hbase in various attributes of relational and distributed database system principles. Detailed study and analysis of architecture and internal working cassandra, Mongo DB and HBase is done theoretically and core concepts are depicted. This paper also presents evaluation of cassandra for an industry specific use case and results are published.

KEYWORD

Nosql, distributed database, cassandra, Mongo DB, Hbase, comparative study of nosql databases

1. INTRODUCTION

Web scaling is contributed by millions of concurrent internet users and biggest web applications generating huge amount of complex and unstructured data, posing uncertainty over traditional relational database management systems to handle enormous volumes of users and data coined as Big data. Era dominated by relational databases has slightly given way to the emergence of nosql technologies which follow distributed database system model to make systems scale easily and to handle large volume of users and data.

1.1 Distributed databases

In distributed database system models, data is logically integrated ; data storage and processing is physically distributed across multiple nodes in a cluster environment. Distributed database is a collection of multiple nodes connected together over a computer network and act as a single point to the user. Advantages of distributed database system include hardware scalability, replication of data across nodes in the cluster, concurrent transactions, availability in case of node failure database will still be online and performance improvements because of distributed hardware.

1.2 Nosql Movement

Relational databases are defined by ACID properties in a system

Atomicity : All of the operations in the transaction will complete, or none will.
Consistency : Transactions never observe or result in inconsistent data.
Isolation : The transaction will behave as if it is the only operation being performed
Durability : Upon completion of the transaction, the operation will not be reversed.

The increasing amount of data in the web is a problem which has to be considered by successful web pages like the ones of Facebook, Amazon and Google. Besides dealing with tera and petabytes of data, massive read and write requests have to be responded without any noticeable latency. In order to deal with these requirements, these companies maintain clusters with thousands of commodity hardware machines. Due to their normalized data model and their full ACID support, relational databases are not suitable in this domain, because joins and locks influence performance in distributed systems negatively. In addition to high performance, high availability is fundamental requirement of many companies. Therefore, databases must be easily replicable and have to provide an integrated failover mechanism to deal with node or datacenter failures. They also must be able to balance read requests on multiple slaves to cope with access peaks which can exceed the capacity of a single server. Since replication techniques offered by relational databases are limited and these databases are typically based on consistency instead of availability, these requirements can only be achieved with additional effort and high expertise. Due to these requirements, many companies and organizations developed own storage systems, which are now classified as nosql databases.[1]

Nosql is a term often used to describe a class of non-relational databases that scale horizontally to very large data sets but do not in general make ACID guarantees. Nosql data stores vary widely in their offerings and have some distinct features on its own. The CAP Theorem coined by Eric Brewer by 2000 states that it is impossible for a distributed service to be consistent, available, and partition-tolerant at the same instant in time. *Consistency* means that all copies of data in the system appear the same to the outside observer at all times. *Availability* means that the system as a whole continues to operate in spite of node failure. *Partition-tolerance* requires that the system continue to operate in spite of arbitrary message loss. Such an event may be caused by a crashed router or broken network link which prevents communication between groups of nodes. [2]

BASE Basically Available replication and sharding techniques are used in nosql databases to reduce the data unavailability, even if subsets of the data become unavailable for short periods of time. *BASE -- Soft State* ACID systems assume that data consistency is a hard requirement, Nosql systems allow data to be inconsistent and provides options for setting tunable consistency levels. *BASE Consistency* when nodes are added to the cluster while scaling up, need for synchronization arises**,** If absolute consistency is required, nodes need to communicate when read/write operations are performed on a node Consistency over availability.

2. BACKGROUND

2.1 Nosql data stores

2.2.1 Key value data stores

The key-value data stores is simple but are quiet efficient and powerful model. Key value data model stores data in a completely schema free manner. Mechanism is similar to maps or dictionaries where data is addressed by a unique keys and since values are uninterpreted byte arrays, which are completely opaque to the system, keys are the only way to retrieve stored data. The data consists of two parts, a string which represents the key and the actual data which is to be referred as value thus creating a key-value pair. These stores are similar to hash tables where the keys are used as indexes, thus making it faster than RDBMS. The modern key value data stores prefer high scalability over consistency. Hence ad-hoc querying and analytics features like joins and aggregate operations have been omitted. New values of any kind can be added at runtime

without conflicting any other stored data and without influencing system availability. The grouping of key value pairs into collection is the only offered possibility to add some kind of structure to the data model. High concurrency, fast lookups and options for mass storage are provided by key-value stores. Example key value databases include Redis, Memcached, Berkeley DB, Amazon Dynamo DB. Amazon Dynamo DB model provides a fast, highly reliable and cost-effective NOSQL database service designed for internet scale applications It offers low, predictable latencies at any scale [3].

2.2.2 Document store databases

Document store databases refers to databases that store their data in the form of documents. Document stores encapsulate key value pairs within documents, keys have to be unique. Every document contains a special key "ID", which is also unique within a collection of documents and therefore identifies a document explicitly. In contrast to key value stores, values are not opaque to the system and can be queried as well. Documents inside a document-oriented database are somewhat similar to records in relational databases, but they are much more flexible since they are schema less. The documents are of standard formats such as XML, PDF, JSON etc. In relational databases, a record inside the same database will have same data fields and the unused data fields are kept empty, but in case of document stores, each document may have similar as well as dissimilar data. Documents in the database are addressed using a unique *key* that represents that document. Storing new documents containing any kind of attributes can as easily be done as adding new attributes to existing documents at runtime. The most prominent document stores are CouchDB , MongoDB, Ravendb. CouchDB and RavenDB do in fact store their data in JSON. MongoDB uses a twist on JSON called Binary JSON (BSON) that's able to perform binary serialization. Document oriented databases should be used for applications in which data need not be stored in a table with uniform sized fields, but instead the data has to be stored as a document having special characteristics document stores should be avoided if the database will have a lot of relations and normalization [1][3].

2.2.3 Column family data stores

Column Family Stores are also known as column oriented stores, extensible record stores and wide columnar stores. All stores are inspired by Google's Bigtable which is a distributed storage system for managing structured data that is designed to scale to a very large size. Column stores in nosql are actually hybrid row/column store unlike pure relational column databases. Although it shares the concept of column-by-column storage of columnar databases and columnar extensions to row-based databases, column stores do not store data in tables but store the data in massively distributed architectures. In column stores, each key is associated with one or more attributes (columns). A Column store stores its data in such a manner that it can be aggregated rapidly with less I/O activity. It offers high scalability in data storage. The data which is stored in the database is based on the sort order of the column family.[3] Columns can be grouped to column families, which is especially important for data organization and partitioning Columns and rows can be added very flexibly at runtime but column families have to be predefined oftentimes, which leads to less flexibility than key value stores and document stores offer. Examples of column family data stores include Hbase, Hypertable, cassandra.

2.2.4 Graph databases

Graph databases are specialized on efficient management of heavily linked data. Therefore, applications based on data with many relationships are more suited for graph databases, since cost intensive operations like recursive joins can be replaced by efficient traversals. Ne04j and GraphDB are based on directed and multi relational property graphs. Nodes and edges consist of objects with embedded key value pairs. The range of keys and values can be defined in a schema, whereby the expression of more complex constraints can be described easily. The range of keys

and values can be defined in a schema, whereby the expression of more complex constraints can be described easily. Therefore it is possible to define that a specific edge is only applicable between a certain types of nodes. Twitter stores many relationships between people in order to provide their tweet following service. Use cases for graph databases are location based services, knowledge representation and path finding problems raised in navigation systems, recommendation systems and all other use cases which involve complex relationships. Property graph databases are more suitable for large relationships over many nodes, whereas RDF is used for certain details in a graph. FlockDB is suitable for handling simple I-hop-neighbor relationships with huge scaling requirements. [3]

3. CASSANDRA

Apache cassandra in a nutshell is an open source, peer to peer distributed database architecture, decentralized, easily scalable, fault tolerant, highly available, eventually consistent, schema free, column oriented database. Generally in a master/slave setup, the master node can have far-reaching effects if it goes offline. By contrast, cassandra has a peer-to-peer distribution model, such that any given node is structurally identical to any other node—that is, there is no "master" node that acts differently than a "slave" node. The aim of cassandra's design is overall system availability and ease of scaling. *cassandra data model* comprises of Keyspace (something like a database in relational databases) and column families (tables). cassandra defines a column family to be a logical division that associates similar data. Basic cassandra data structures: the column, which is a name/value pair and a client-supplied timestamp of when it was last updated, and a column family, which is a container for rows that have similar, but not identical, column sets. There is no need to store a value for every column every time a new entity is stored. For example, column family data model looks like figure 1. A cluster is a container for keyspaces—typically a single keyspace. A keyspace is the outermost container for data in cassandra, but it's perfectly fine to create as many keyspaces as the application needs. A *column family* is a container for an ordered collection of rows, each of which is itself an ordered collection of columns.[4]

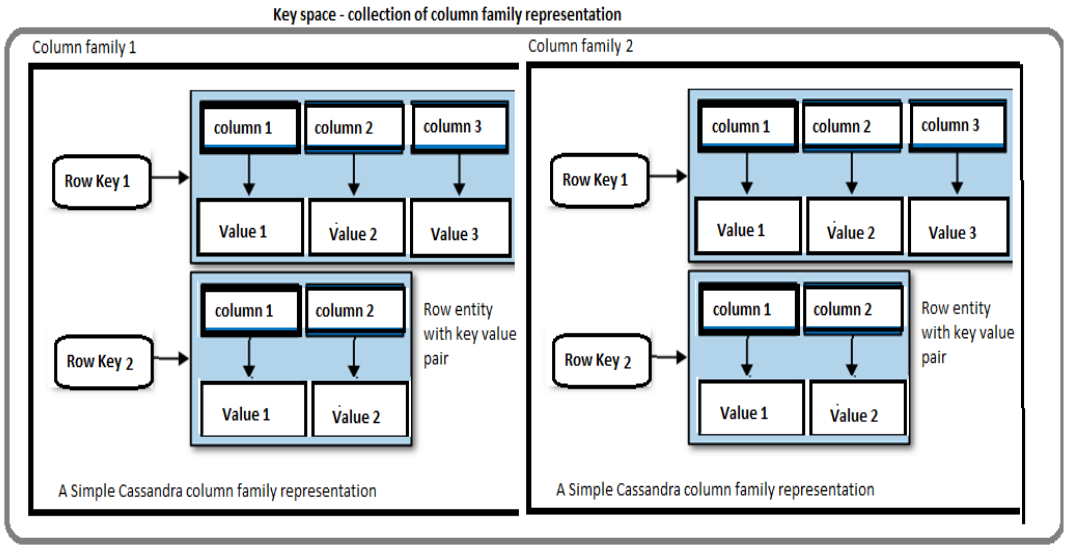

Figure 1. cassandra data model

Replication is set by replication factor which defines how many copies of each piece of data will be stored and distributed throughout the cassandra cluster. With a replication factor of one, data

will exist only in a single node in the cluster. Losing that node means that data becomes unavailable. It also means that cassandra will have to do more work as coordinator among nodes; if all the data for a given key. Higher replication factor indicates higher availability of cluster which is ideal, and replication factor can never be set than a value greater than the number of nodes present. SimpleStrategy places the first replica on a node determined by partitioner. Additional replicas are placed on the next nodes clockwise in the ring without considering rack or data center location. NetworkTopologyStrategy is used to have cluster deployed across multiple data centers. With replication and peer to peer model cassandra is fault tolerant and provides no single point of failure.

Partitioning defines how data will be distributed across the cassandra nodes and allow you to specify how row keys should be sorted, which has a significant impact on the options available for querying ranges of rows. Random partitioner with an MD5 hash applied to it to determine where to place the keys on the node ring. This has the advantage of spreading your keys evenly across your cluster, because the distribution is random. It has the disadvantage of causing inefficient range queries. Order preserving partitioner, the token is a UTF-8 string, based on a key. Rows are therefore stored by key order, aligning the physical structure of the data with your sort order.[4]

Cassandra uses a gossip protocol for intra-ring communication so that each node can have state information about other nodes. The gossiper runs every second on a timer. Hinted handoff is triggered by gossip, when a node notices that it has hints for a node just came back online. Hints are recorded when a node in the cluster went offline all the changes supposed to be on the offline node will be marked as hints and replayed once the node is back online. Gossip protocol is deployed in distributed systems wherein it acts as an automatic mechanism for cluster communication, failure detection and replication. Gossip protocol sends Heartbeat signals every one second across the distributed cluster to maintain list of active and dead nodes. Anti-Entropy is the replication synchronization mechanism used in cassandra to ensure replicas across the cluster are updated to the latest version of data.

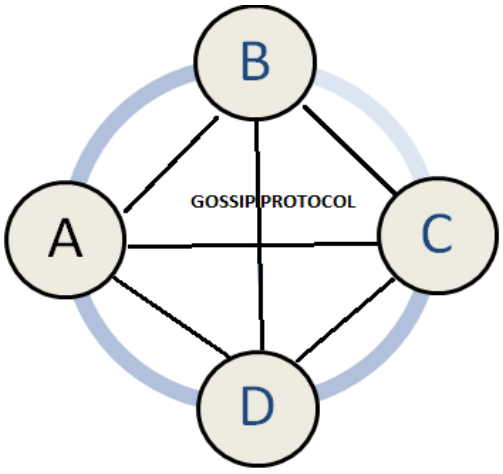

Figure 2. Cassandra cluster communication

Level	Basic Consistency level description
ONE	A write must be written to the commit log and memory table of at least one replica node.
QUORUM	A write must be written to the commit log and memory table on a quorum of replica nodes.
ALL	A write must be written to the commit log and memory table on all replica nodes in the cluster for that row key.
ANY	A write must be written to at least one node.

Table 1. cassandra consistency

Durability in cassandra is ensured with the help of commit logs, which a crash recovery mechanism. Writes will not be considered successful until data is written to commit logs to support durability goals. After it's written to the commit log, the value is written to a memory-resident data structure called the *memtable*. When the number of objects stored in the memtable reaches a threshold, the contents of the memtable are flushed to disk in a file called an SSTable. A new memtable is then created. Once a memtable is flushed to disk as an SSTable, it is

immutable and cannot be changed by the application. Despite the fact that SSTables are compacted, this compaction changes only their on-disk representation.

Writes are very fast in cassandra, because its design does not require performing disk reads or seeks. The memtables and SSTables save cassandra from having to perform these operations on writes, which slow down many databases. All writes in cassandra are append-only. Because of the database commit log and hinted handoff design, the database is always writeable, and within a column family, writes are always atomic. cassandra's best feature is tunable consistency levels which lets user specify consistency level based on the requirements. A higher consistency level means that more nodes need to respond to the query, giving you more assurance that the values present on each replica are the same. If two nodes respond with different timestamps, the newest value wins, and that's what will be returned to the client. In the background, cassandra will then perform what's called a *read repair* it takes notice of the fact that one or more replicas responded to a query with an outdated value, and updates those replicas with the most current value so that they are all consistent.

cassandra is often communicated as being an eventually consistent data store. It does so by requiring that clients specify a desired consistency level– zero, one, quorum, all, or any with each read or write operation. Use of these consistency levels should be tuned in order to strike the appropriate balance between consistency and latency for the application. In addition to reduced latency, lowering consistency requirements means that read and write services remain more highly available in the event of a network partition. A consistency level of zero indicates that a write should be processed completely asynchronously to the client. A consistency level of one means that the write request won't return until at least one server where the key is stored has written the new data to its commit log. A consistency level of all means that a write will fail unless all replicas are updated durably. quorum requires that (N/2 + 1) servers must have durable copies where N is the number of replicas [2]. A write consistency of any has special properties that provide for even higher availability at the expense of consistency. Read and write consistency levels can be set to different values based on the requirements. cassandra differs from many data stores in that it offers much faster write performance than read performance. There are two settings related to how many threads can perform read and write operations: concurrent_reads and concurrent_writes can be configured for concurrency.

cassandra uses its own *CQL cassandra query language* to interact with its column family data model. cassandra unlike RDBMS has no referential integrity constraint and no joins indeed. cassandra performs best when the data model is denormalized. There is no first-order concept of an update in cassandra, meaning that there is no client query called an "update." An insert statement for a key that already exists, cassandra will overwrite the values for any matching columns; if your query contains additional columns that don't already exist for that row key, then the additional columns will be inserted so no duplicate keys are possible. cassandra automatically gives you record-level atomicity on every write operation. In RDBMS, row-level locking has to be specified. Although cassandra offers atomicity at the column family level, it does not guarantee isolation and no locks.

4. MONGO DB

MongoDB is a flexible and scalable document oriented data store with dynamic schemas, auto-sharding, built-in replication and high availability, full and flexible index support, rich queries, aggregation. It combines the ability to scale out with many of the most useful features of relational databases, such as secondary indexes, range queries, and sorting. Mongo DB follows a master/slave approach, and it has a automatic failover feature where if a master server goes down, MongoDB can automatically failover to a backup slave and promote the slave to a master. Mongo DB data model consists of *document* which is the basic unit of data for MongoDB equivalent to a row in a relational database management system. Grouping of similar documents

is called a *collection* can be thought of as the schema-free equivalent of a table. A single instance of MongoDB can host multiple independent databases, each of which can have its own collections and permissions similar to relational databases. Every document has a special key "_id" which is unique across the document's collection. Mongo DB document store contains references to store the relationships between data by including links or references from one document to another. Applications can resolve these references to access the related data. These are normalized data models. Embedded documents capture relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures as sub-documents in a field or array within a document.[6] These denormalized data models allow applications to retrieve and manipulate related data in a single database operation. Mongo DB collection can consists of simple documents, with some reference documents embedded documents also as shown in figure

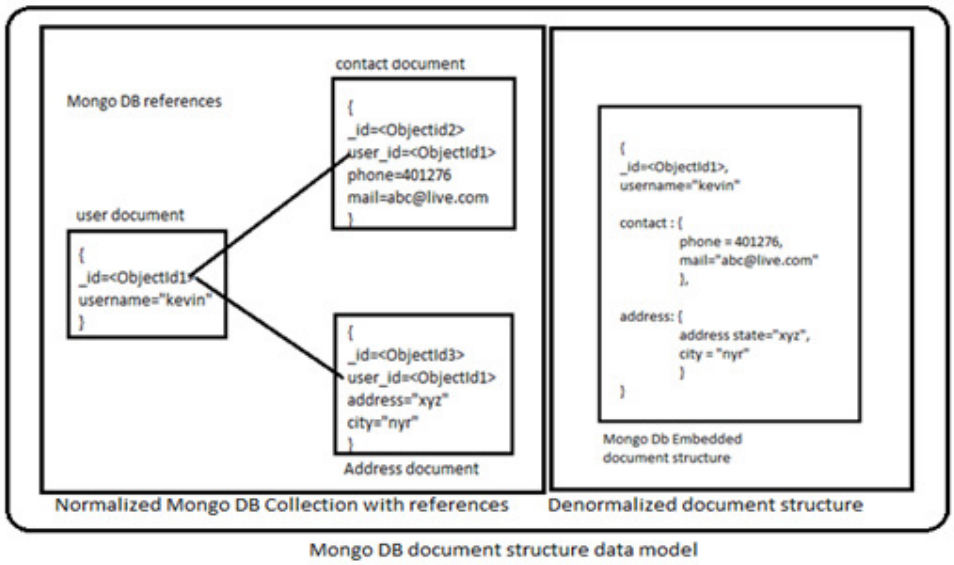

Figure 3. Mongo Db data model

Master-slave replication is the most general replication mode supported by MongoDB, very flexible for backup, failover, read scaling. A *replica set* is basically a master-slave cluster with automatic failover. Major difference between a master-slave cluster and a replica set is that a replica set does not have a single master: one is elected by the cluster and may change to another node if the current master goes down. However, they look very similar: a replica set always has a single master node (called a *primary*) and one or more slaves (called *secondaries*). If the current primary fails, the rest of the nodes in the set will attempt to elect a new primary node. This election process will be initiated by any node that cannot reach the primary. The new primary must be elected by a majority of the nodes and with the highest priority in the set. The primary node uses a heartbeat to track how many nodes in the cluster are visible to it. If this falls below a majority, the primary will automatically fall back to secondary status thus automatic failover happens in Mongo DB. The primary purpose and most common use case of a MongoDB slave is to function as failover mechanism in the case of data loss or downtime on the master node. Other valid use cases for a MongoDB slave can be used as a source for taking backups and slaves can be used to serve requests to reduce load on master.

Sharding is MongoDB's approach to scaling out. Sharding allows you to add more machines to handle increasing load and data size horizontally without affecting your application. *Sharding* refers to the process of splitting data up and storing different portions of the data on different machines; the term *partitioning* is also sometimes used to describe this concept. By splitting data up across machines, it becomes possible to store more data and handle more load without

requiring large or powerful machines. MongoDB supports *autosharding*, which eliminates some of the administrative headaches of manual sharding. The cluster handles splitting up data and rebalancing automatically. The basic concept behind MongoDB's sharding is to break up collections into smaller *chunks*. These chunks can be distributed across *shards* so that each shard is responsible for a subset of total data set. *Mongod* is the Mongo DB database instance that should be initiated and running on the servers that hold data or shard. For *range-based sharding*, MongoDB divides the data set into ranges determined by the shard key. For *hash based partitioning*, MongoDB computes a hash of a field's value.

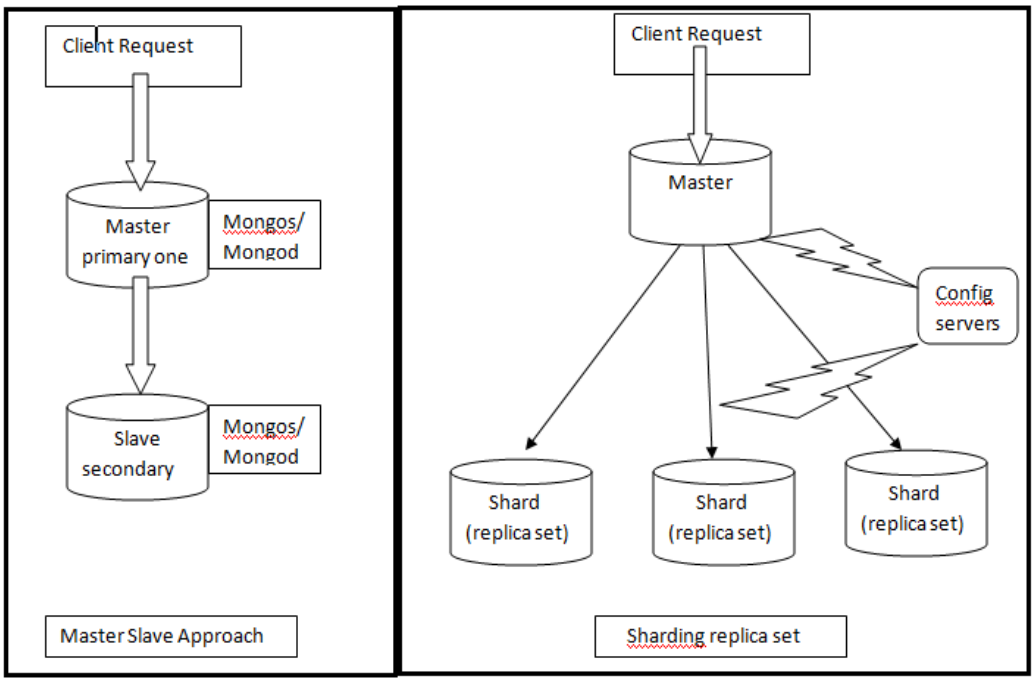

Figure 4. Mongo DB Cluster Models

Application doesnot know which shard has what data, or even that our data is broken up across multiple shards, so there is a routing process called mongos in front of the shards. This router knows where all of the data is located, so applications can connect to it and issue requests normally. The router, knowing what data is on which shard, is able to forward the requests to the appropriate shard(s). If there are responses to the request, the router collects them and sends them back to the application.[5] When sharding is setup, a key is choosen from a collection and use that key's values to split up the data. This key is called a *shard key*. Sharding basically involves three different components working together: A shard is a container that holds a subset of a collection's data. A shard is either a single mongod server or a replica set. Mongos is the router process routes requests and aggregates responses. Config servers store the configuration of the cluster: which data is on which shard. Because mongos doesn't store anything permanently, it needs to get the shard configuration. It syncs this data from the config servers.

The master keeps a record of all operations that have been performed on it. The slave periodically polls the master for any new operations and then performs them on its copy of the data. The record of operations kept by the master is called the *oplog*, short for operation log. Each document in the oplog represents a single operation performed on the master server. The documents contain several keys, including the following: ts Timestamp for the operation. *op* type of operation performed, *ns* collection name where the operation was performed, *o* document further specifying the operation to perform thus ensures durability of Mongo DB. MongoDB uses

write ahead logging to an on-disk journal to guarantee durability and to provide crash resiliency. Before applying a change to the data files, MongoDB writes the change operation to the journal.

Write concern describes the guarantee that MongoDB provides when reporting on the success of a write operation. The strength of the write concerns determine the level of guarantee. When inserts, updates and deletes have a *weak* write concern, write operations return quickly. In some failure cases, write operations issued with weak write concerns may not persist. With *stronger* write concerns, clients wait after sending a write operation for MongoDB to confirm the write operations. MongoDB provides different levels of write concern to better address the specific needs of applications. For sharded collections in a shared cluster, mongos directs write operations from applications to the shards that are responsible for the specific *portion* of the data set. The mongos uses the cluster metadata from config servers to route the write operation to the appropriate shards. Read preference describes how MongoDB clients route read operations to members of a replica set. By default, an application directs its read operations to the primary member in a replica set. Reading from the primary guarantees that read operations reflect the latest version of a document. However, by distributing some or all reads to secondary members of the replica set, you can improve read throughput or reduce latency for an application that does not require fully up-to-date data. [6]

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must scan every document in a collection to select those documents that match the query statement. Indexes in Mongo DB are similar to relational databases and there are many types of indexes top support. The disadvantage to creating an index is that it puts a little bit of overhead on every insert, update, and remove. This is because the database not only needs to do the operation but also needs to make a note of it in any indexes on the collection. Thus, the absolute minimum number of indexes should be created. MongoDB provides a number of aggregation operations that perform specific aggregation operations on a set of data such as count, distinct, group. MongoDB documents are BSON documents. BSON is a binary representation of JSON with additional type information. In the documents, the value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents. Data modification refers to operations that create, read, update, or delete data commonly known as CRUD operations. In MongoDB, these operations modify the data of a single collection. For the update and delete operations, criteria can be specified to select the documents to update or remove. Insert(), update(), delete() java script methods are used for data modification operations.

5. Hbase

Apache HBase is an open source, non-relational, persistent, strictly consistent fault tolerant distributed database runs on top of HDFS (Hadoop Distributed File System)modeled after Google's Big table providing the capabilities on hadoop. Hbase is a Master/Slave approach comprising of one master server and many region servers where the master node is responsible for assigning or load balancing across region servers. Region servers are slaves like responsible for all read and write requests for all regions they serve, and also split regions that have exceeded the configured region size thresholds. The store files are typically saved in the Hadoop Distributed File System (HDFS), which provides a scalable, persistent, replicated storage layer for HBase. It guarantees that data is never lost by writing the changes across a configurable number of physical servers.

Hbase follows Big table data model, a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterrupted array of bytes. Hbase data model is column oriented storage structure typically grouped into one or more tables. Row keys in table are arbitrary strings; Bigtable maintains data in lexicographic order by row key. The row range for a table is dynamically partitioned. Column keys are grouped into sets called *column families*, which form the basic unit of access control. All

data stored in a column family is usually of the same type. A column family must be created before data can be stored under any column key in that family;[7] Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by 64-bit integers timestamp. Although conceptually a table is a collection of rows with columns in HBase, physically they are stored in separate partitions called *regions*. Every region is served by exactly one region server, which in turn serves the stored values directly to clients.

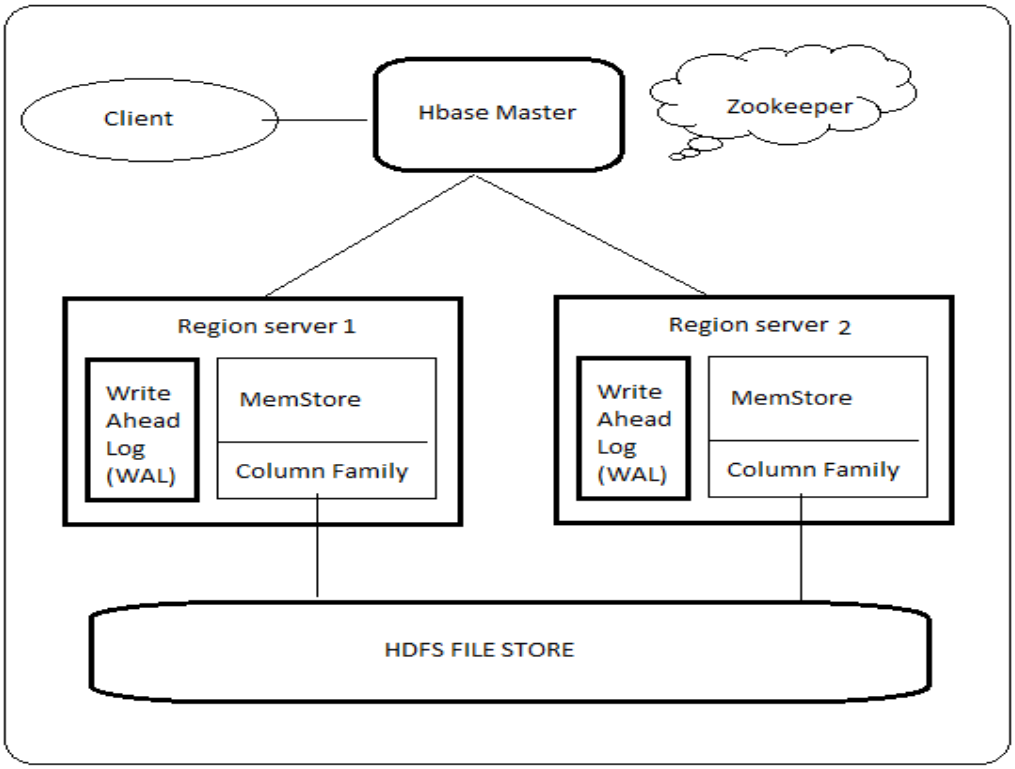

Figure 5. Hbase data model

Architecture consists of three major components to HBase: the client library, one master server, and many region servers. The HMaster in the HBase is responsible for performing administration, managing and monitoring the cluster, assigning regions to the region servers and controlling the load balancing and failover. The HRegionServer performs hosting and managing regions, splitting the regions automatically, handling the read/write requests and communicating with the clients directly. The region servers can be added or removed while the system is up and running to accommodate changing workloads. The master is responsible for assigning regions to region servers and uses *Apache ZooKeeper*, a reliable, highly available, persistent and distributed coordination service, to facilitate the task. ZooKeeper is the comparable system to Google's use of Chubby for Bigtable. It offers filesystem-like access with directories and files distributed systems can use to negotiate ownership, register services, or watch for updates. Every region server creates its own ephemeral node in ZooKeeper, which the master, in turn, uses to discover available servers. They are also used to track server failures or network partitions. Ephemeral nodes are bound to the session between ZooKeeper and the client which created it. The session has a heartbeat keepalive mechanism that, once it fails to report, is declared lost by ZooKeeper and the associated ephemeral nodes are deleted. HBase uses ZooKeeper also to ensure that there is only one master running, to store the bootstrap location for region discovery, as a registry for region servers, as well as for other purposes. ZooKeeper is a critical component, and without it HBase is not operational. [8]

Hbase replication is differnet from the cassandra and Mongo DB, because Hbase is tightly coupled with Hadoop Distributed file system. HBase *replication* enables to have multiple clusters that ship local updates across the network so that they are applied to the remote copies. Replication scope determines enabling and disabling of replication in Hbase. By default, replication is disabled and the *replication scope* is set to 0, Setting replication scope to 1 enables replication to remote clusters. Default replication factor of HDFS is 3 hence if you create a HBase table and put some data on it, the data is written on HDFS and three copies of that data are created. Hbase is built on top of HDFS, which provides replication for the data blocks that make up the Hbase tables. All data writes in HDFS go to the local node first, if possible, another node on the same rack, and another node on a different rack (given a replication factor of 3 in HDFS). Hbase supports auto-sharding feature for scalability and load balancing in HBase. Regions are essentially contiguous ranges of rows stored together, dynamically split by the system when they become too large. Alternatively, they may also be merged to reduce their number and required storage files. HBase regions are equivalent to *range partitions* as used in database sharding and can be spread across many physical servers, thus distributing the load, and therefore providing scalability and fault tolerance.

Hbase communication flow is that a client contacts the ZooKeeper first when trying to access a particular row. It does so by retrieving the server name and the metadata information required to access region servers and fetch the results. Hbase has two file formats, one for Write ahead Log(WAL) and other file is actual data storage file. When there are writes to Hbase Write ahead Log is the first place where the data is written to. Once the data is written to the WAL, it is placed in the memstore and it will check to see if memstore is full a flush to disk is requested. The store files are monitored by a background thread to keep them under control. The flushes of memstores slowly build up an increasing number of on-disk files. If there are enough of them, the *compaction* process will combine them to a few, larger files. This goes on until the largest of these files exceeds the configured maximum store file size and triggers a region split. Writes are written to Write Ahead log, and only if the update has succeeded is the client informed that the operation has succeeded. The master and region servers need to orchestrate the handling of logfiles carefully, especially when it comes to recovering from server failures. The WAL is responsible for retaining the edits safely; replaying the WAL to restore a consistent state thus Hbase ensures *durability*. Hbase follows *strict consistency model*, writes are written to single master, on CAP theorm Hbase focuses on Consistency and partition tolerance, offering strict consistency model for optimized reads. Hbase works very well on HDFS platform and map reduce can be very effective for bulk loading and read operations.

6. COMPARATIVE STUDY OF CASSANDRA, MONGO DB AND Hbase

PARAMETER	CASSANDRA	MONGODB	HBASE
Nosql classification	Column family databases	Document store database	Column family database on HDFS
Architecture	Peer to peer architecture model	1.master slave 2.peer to peer via sharding	Master Slave architecture model

Consistency	Tunable Consistency. Read and write consistency levels can be set	Tunable consistency. Write concern and read preference parameters can be configured.	strict consistency (focuses mainly on consistency according to cap theorem).
Availability	Very high availability (focuses mainly on availability according to cap theorem)	High availability with help of sharding	Failover clustering to provide availablity in case of master node failure.
Partitioning	Supports partitioning (random partitioner, byteorder partitioner)	Sharding supports partitioning range and hash based. Auto-sharding is built-in feature	Hbase regions provides range partitioning.
Data Model	Keyspace - columnfamily	Collection-document	Regions-column family
Replication	Replication strategy can be defined by setting Replication Factor	Configurable replica set for Mongo DB replication	Hbase has Replication scope (0- disabled 1-enabled). HDFS has replication factor
Fault Tolerance	No single point of failure with peer to peer architecture	No single point of failure with sharding approach as we can configure multiple mongo s instances. Single point of failure in master slave approach.	Single point of failure in master slave approach. Can be overcome by failover clustering.
Cluster Communication	cassandra uses gossip protocol for inter node communication	Mongos instances are configured to route requests from master to slave nodes	Apache Zookeeper is responsible for Hbase node co-ordination.
Writes	Very fast writes	Fast when data is in RAM	Writes slower than

performance	because of peer to peer architecture and cassandra data model	and latency increases for huge amount of data, very fast writes if in memory writes with allowance for data loss	cassandra if it uses pipelined writes (synchronous). Asynchronous writes are configurable
Reads performance	Performance based on consistency level (decreases in performance with increase in consistency level) and replication factor.	In a master/slave setup, any changes are written to the master and then passed on to slaves. This model is optimized for reading data, as it allows data to be read from any slave. In sharding reads depend on eventual/strict consistency level.	Follows strict consistency model and are optimized for reads. Very fast reads in Hbase with Hadoop support.
Durability	Achieved using a commit log	Achieved using write ahead logging. However, if in memory writes than durability is not guaranteed.	Achieved using Write Ahead Log (WAL)
Concurrency	Row level locking.	No concurrency for write operations. Database level (global) locks for each write.	Row level locking.
Aggregate Functions	No support for aggregate and group by functions	Supports aggregate functions by default	Supports aggregate functions via hive.
Indexing techinque	Hash indexes	B tree indexes. Facilitate better performance for range queries	LSM trees that are similar to b trees
Map Reduce	Can support map reduce integration with Hadoop.	Has Map reduce by default.	Very good support for map reduce because of HDFS.

Table 2. Comparative study of cassandra, Mongo DB, Hbase

7. EVALUATION OF CASSANDRA

Applications generating data has increased in huge volumes in this internet era. An industry specific use case for nosql database solution is discussed here. Tracking the user activity of applications with relational databases is becoming tedious, as they generate many GB's of log data every day. The ultimate need for this analysis and comparative study was to come up with a nosql database solution for user Activity logging in production Environment. Existing logging mechanism in production environment uses relational database and has its known problems in scalability and storing unstructured data. So the idea was to have an effective logging solution using nosql data stores which promises high scalability, high availability and fault tolerance.

An industry specific effective logging solution requires minimum performance hit, less storage space, zero downtime, ability to scale easily, distributed system supporting parallel operations, very high write performance, highly reliable and concurrent. Based on the above comparative study Table 2 of various features of cassandra, Mongo DB and Hbase with its peer to peer architecture model, high availability, tunable consistency, very fast write performance and high concurrency cassandra seems to best fit the mentioned industrial use case to have an effective activity logging solution. Evaluation of cassandra for performance and concurrency in comparison with relational databases its test cases and results are published here.

Hardware specifications for test cases had 3-node cassandra cluster for POC with each machine Memory:1GB,DiskSpace:100GB,OS:Centos5.7,Java version 1.6.0_43 configuration. Test cases are primarily focused on cassandra's write performance and concurrency required for logging systems. Below are the results for cassandra write performance and currency , read performance in comparison with relational databases RDBMS.

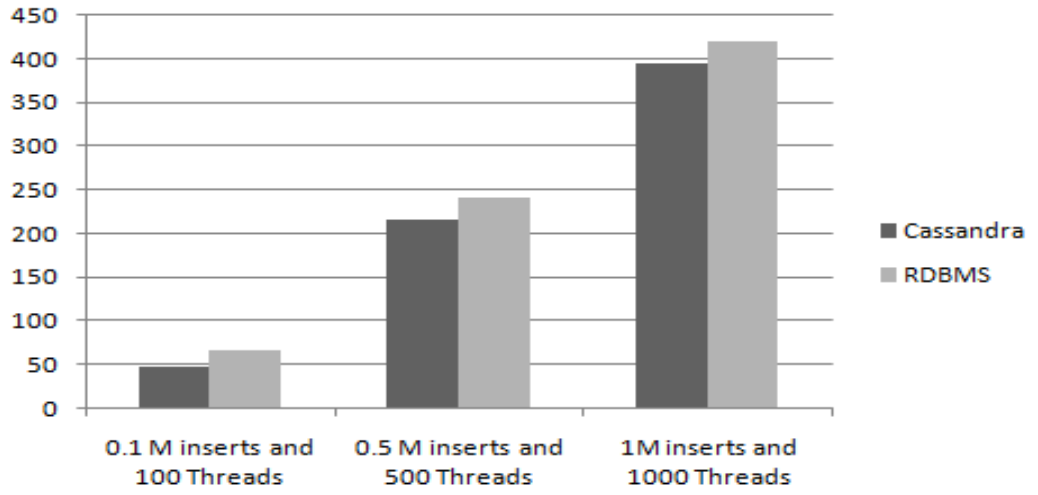

Figure 6. Write performance and concurrency

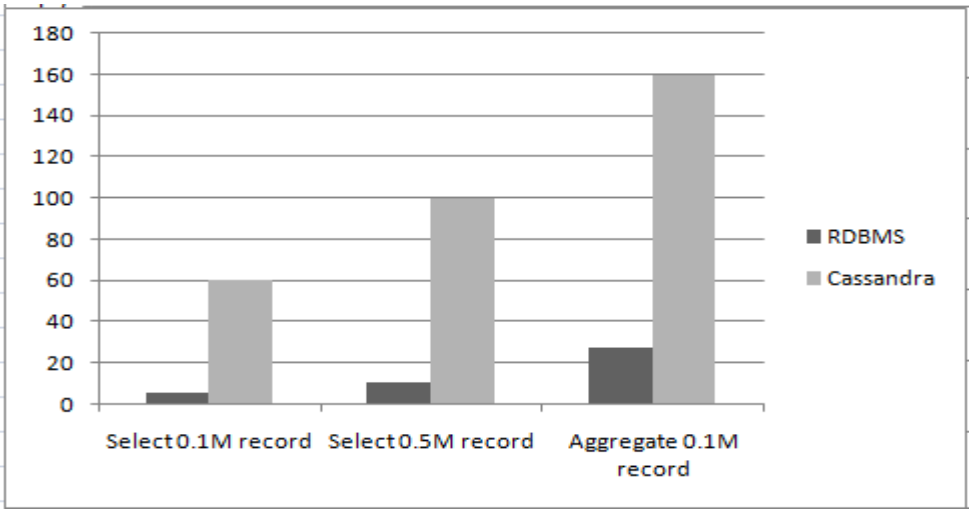

Figure 7. Read performance

7.1 Test cases and observations

1. Write performance of cassandra 3 node cluster with testcases of different concurrency levels are shown in write performance and currency figure. Write performance of cassandra is very fast. 2000 inserts per sec compared to RDBMS 1500 inserts/sec.

2. cassandra was tested with 100,500 and 1000 concurrent threads and the cluster write performance was good. High levels of concurrency (1000-2000 plus concurrent threads can hit the cassandra cluster at the same time without any failure).

3. However read performance of cassandra to RDBMS is hugely in favour of RDBMS and read performance of cassandra is slower.

4. Load testing of cassandra was done with 100 threads inserting continuous data for continuous 20 hours (1 day approx). No issues seen passed stress testing.

5. Test cases for replication, fault tolerance, tunable consistency, compression were also satisfactory.

8. CONCLUSION AND FUTURE WORK

Nosql databases are not "One size fits all". Each nosql classification addresses a specific data storage and processing requirements. cassandra, Mongo DB and HBase are popular among nosql databases and a detailed comparative study is made to understand their features and working. cassandra can be used for applications requiring faster writes and high availability. Mongo DB fits for usecases with document storage, document search and where aggregation functions are mandate. Hbase suits the scenarios where hadoop map reduce is useful for bulk read and load operations hbase offers optimized read performance with hadoop platform.

Currenlty working on performance evaluation of Cassandra, Mongo db and Hbase in the aspects of read and write performance, consistency levels and indexing sharding performance with Map reduce. Future work will encompass the performance analysis results and comparisons.

REFERENCES

[1] Robin Hecht Stefan Jablonski, University of Bayreuth " NoSQL Evaluation A Use Case Oriented Survey" 2011 International Conference on Cloud and Service Computing

[2] Dietrich Featherston "cassandra: Principles and Application" Department of Computer Science University of Illinois at Urbana-Champaign

[3] Ameya Nayak, Anil Poriya Dept. of Computer Engineering Thakur College of Engineering and Technology University of Mumbai " Type of NOSQL Databases and its Comparison with Relational Databases" International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868 Foundation of Computer Science FCS, New York, USA Volume 5– No.4, March 2013

[4] Eben Hewitt Apache cassandra project chair "cassandra the definitive guide" Published by O'Reilly Media November 2010

[5] Kristina Chodorow and Michael Dirolf "Mongo DB: the definitive guide" Published by O'Reilly Media September 2010

[6] http://www.mongodb.org/

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber "Bigtable: A Distributed Storage System for Structured Data" Google, Inc.

[8] Lars George "Hbase the definitive guide" Published by O'Reilly Media September 2011

[9] Pokorny, J." Nosql databases: a step to database scalability in web environment" Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services. pp. 278 283. iiWAS '11, ACM, NewYork, NY, USA (2011)

[10] Elif Dede, Bedri Sendir, Pinar Kuzlu, Jessica Hartog, Madhusudhan Govindaraju Grid and Cloud Computing Research Laboratory SUNY Binghamton, New York, USA "An Evaluation of Cassandra for Hadoop" IEEE Cloud 2013

[11] http://www.datastax.com/

[12] http://hbase.apache.org/

[13] Philippe Cudr_e-Mauroux1, Iliya Enchev1, Sever Fundatureanu2, Paul Groth2, Albert Haque3, Andreas Harth4, Felix Leif Keppmann4, Daniel Miranker3, Juan Sequeda3, and Marcin Wylot1,1 University of Fribourg, 2VU University Amsterdam, 3University of Texas at Austin, 4 Karlsruhe Institute of Technology " NoSQL Databases for RDF:An Empirical Evaluation".

[14] Urbani, J., Kotoulas, S., Maassen, J., Drost, N., Seinstra, F., Harmelen, F.V.,Bal, H.: H.: Webpie: A web-scale parallel inference engine. In: In: Third IEEE International Scalable Computing Challenge (SCALE2010), held in conjunction with the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) (2010)

[15] Morsey, M., Lehmann, J., Auer, S., Ngomo, A.C.N.: Dbpedia sparql benchmark{ performance assessment with real queries on real data. In: The Semantic Web{ ISWC 2011, pp. 454{469. Springer (2011)