NoSQL Databases

[Alankritha Yata]

1. Introduction

NoSQL databases provide storage and retrieval of data which is designed differently from tabular relations used in relational databases. NoSQL also called as Not Only SQL is an approach to a data design that is useful for large datasets of distributed data. Initially, the NoSQL databases came into existence to modernize web scale databases. It was during early 2009, they came into the picture and started to grow rapidly because of its support to schemaless, eventually consistent and simple API. When we say Not Only SQL it doesn't mean it is against RDBMS or has no features of RDBMS. It simply avoids certain functionalities of RDBMS databases and replaces it with its own functionality. Consider tables for this example. Though NoSQL has tables, their schema is much more flexible. It doesn't use SQL as a language to manipulate data. NoSQL databases are not full ACID (atomicity, consistency, isolation and durable). But they still have the ability to deal with fault tolerance. Similar to ACID, it supports BASE property meaning, Basically available, Soft state, Eventual consistency. The names itself explain their functionality. When an organization needs to work on massive amounts of unstructured data which is remotely stored on the cloud and analyze it, the best solution will be NoSQL[1]. It has the capability to scale the humongous amount of data and still manage the data performance.

All the databases are developed with the inbuilt capability of Scaling and big data performance. Scaling in this context means horizontal scaling. The capability to enlarge the databases upon requirement is called scaling. There are two types of scaling. They are horizontal scaling and vertical scaling. Horizontal scaling simply adds machines when there is a requirement wherein vertical scaling tries to increase the size of one machine when required. NoSQL is being attracted because of its horizontal scalability feature. NoSQL is assumed to be as a modern database choice because of its ability to scale with web requirements. NoSQL databases are designed by observing some class of problems faced by RDBMS and cannot be dealt with. Those class of problems is being more flexible about stored data, aggregating data and targeting use case like relationships or simplifying data. Though they are relatively new and are based on unproven technology, many big organizations are using them for various purposes. Few Organizations use various NoSQL depending on their requirement. For example, they might use Mongo DB for replication, Cassandra for storage etc.

Relational databases like SQL has been the first priority for database management. But from past few decades unstructured query language has been gaining importance as an alternative model for database management. There are a lot of factors responsible for this reason which will be discussed further. Structured query programming language is used to communicate with a relational database where you can manage, retrieve and store data. SQL[1] is the language to manipulate the data. It had all features to call it a perfect database supporting transactions, query processing, and storage management. Gradually RDBMS has lost its importance because of the increasing amount of data. Data has grown so much that segregating this, storing this in the tables every now and then was decreasing the performance and increasing the cost. The reason behind this is vertical scalability meaning, it tries to increase the storage in one machine.

Trying to increase the storage every time it is required is a very expensive task. This is when the NoSQL databases have gained the prominence. It has the ability to organize the humongous amount, of data without slowing

down the performance. The database is schemaless, meaning we don't have to define the structure before storing the data. We can store the unstructured data without knowing what kind of data you will be collecting and storing. This is constructed on the concept of distributed databases where data is stored across multiple processing servers which help in horizontal scalability. This features of NoSQL databases such as distributed databases, horizontal scalability, and replicating data on servers are responsible for its prominence. The benefits of NoSQL are lesser server cost, schemaless, bigger data handling capability, elastic scaling and maintaining servers at a low cost. Though it has its benefits there are few limitations. There are many key features yet to be implemented in various NoSQL databases. RDBMS vendors provide enterprise support better than NoSQL which provides support to only small start-ups and not bigger companies. Complex queries can be addressed by RDBMS, unlike NoSQL which deals with real-time analytics.

The databases that will be covered in this paper are Aerospike, Oracle NoSQL, Simple DB, RIAK, Cassandra, HBase, BigTable, MongoDB, Couch DB and Amazon Dynamo DB. The first four databases Aerospike, Oracle NoSQL, Simple DB and RIAK fall under key-value stores. This means that their data is stored in the form of key-value pairs. Cassandra, HBase, and BigTable fall under column stores/families and the data here are stored in columns something which is similar to tables, but it is schemaless. Finally, Mongo DB, Couch DB, and Amazon Dynamo belongs to document store databases where data is stored in the form of documents and records. There is one more type called as Graph databases.

The functioning of transaction management, storage management, and query processing play a very important role to understand the aforementioned databases. The above mentioned are not the only NoSQL databases. There are many of them of which the best ones are explained in this paper. The rest of the paper will comprise of analysis sections which explains in details about the types of databases, differences between the databases. Advantages and disadvantages of the databases mentioned and the applications related to these databases including the conclusion will be discussed throughout the paper.

2. Analysis

Before we get in depth about various kinds of NoSQL databases we need to understand CAP theorem. In the process of improving the performance of databases which deal with the humongous amount of data, we had to compromise with consistency, availability, fault tolerance, durability etc. This is explained better with CAP theorem by Eric Brewer which says that any distributed system handling such humongous data cannot guarantee Consistency, Availability and Partition tolerance. Here consistency means, all the nodes see the same data at a time, availability means system remains available all the time and partition tolerance means that in any condition be it the failure of machines or outage the data needs to be available. All the NoSQL databases fall into the category of CAP theorem and no databases satisfy all three properties. We will observe in the further sections about this. As we have discussed above there are four kinds Key-Value store, Document store, Column store and Graph-based.

2.1 Key-Value store

Key-Value store is the simplest NoSQL database as it is schema-less. The value is added, deleted and updated using the key. Here the key can be generated automatically or manually written and the value can be in string format or JSON or BLOB (Basic large object). The notion behind key-value database if the hash table. The keys in the hash table are generated and they are unique. These keys are mapped to values/data using a pointer. It is known that hash table has buckets and each bucket is a collection of keys. We need to know both the bucket and key to read or write a value

into the database. Because of the ease in its working, the complexity of key-value store is trivial.

f we try to reflect back with CAP theorem it is[2] quite evident that Key-value stores support availability and partition aspects but it lacks consistency. Consider the following example where the keys are the names of the students and they are in buckets and the values are as mentioned above. The keys and values are read and written in a database using Get(key) which returns the value associated with the key, Multi-get (key1, key2…keyn) returns a list of values required, put (key, value) allows to write the values and Delete(key) deletes the value associated.

Though Keyvalue store is easy to implement it has its disadvantages like it fails to support any relational databases features and one more disadvantage is the volume of data. It becomes quite a tedious task to maintain unique values for all the keys and it is quite inefficient when there is a lot of querying involved. Among the discussed databases Oracle NoSQL, RIAK, Simple DB, and Aerospike fall under Key-value store. Key-value stores are best suited for applications which have simple data model where there are frequent small reads and writes. Some applications that use key-value stores cache data to improve the performance, storing large objects such as videos and files and storing configuration and user information for mobile applications.

2.1.1 Oracle NoSQL

Oracle NoSQL is built on Berkley DB java platform and it is one of the key-value store databases. It is good for real time implementation and has caught an eye of a lot of companies in the IT industry. The data storage of this database offers good performance and scalability. Oracle offers all the CRUD (Create, Read, Update, and Delete) operations. This especially works well for any web service application which involves interaction with different layers like the web server, application server, and backend databases. Applications make use of the provided KVLite and KVStore.

Features.

Oracle NoSQL is a distributed key-value database. It provides a very simple data model to the application developer. This is built on Berkley DB java platform. This has a key-value pair storage which offers good performance and scalability. It services network requests to store and retrieve data. This database services these type of requests with a latency, throughput and data consistency that is predictable based. It works well for any web service application because it adds a layer of services for use in distributed environments. It also provides transactional semantics for data manipulation and monitoring. This database offers full create, update, read and delete operations with adjustable durability guarantees. It is designed to be highly available while requiring very minimal administration. The features of Oracle NoSQL are it is horizontally scalable, highly available, simple administration and simple key-value data model.

A KvLite version of Oracle NoSQL which is a very simplified version is used. This doesn't need continuous administration. Java –jar lib/kvstore.jar kvlite is the command to start kvlite. Run the KvLite test application using the command java –jar lib/kvclient.jar. Following pictures show an example program running.

Data management.

KVStore is a collection of nodes where the data is spread across all replication nodes. KVStore either takes the

place of your back-end database or runs alongside it.

The store contains several storage nodes as shown in the above figure. A storage node is a physical machine with its own local storage. Every storage node has replication nodes. The number of replication nodes is determined by its own capacity. The database ensures that the store is assigned a load that is proportional to its capacity. Replication node can be thought as a single database which has key-value pairs.

The database has a number of shards and every shard in the database contains partitions. The data is accessed using its key. Once a key is placed in the partition then it cannot be moved to a different partition. This database assigns keys evenly across all available partitions. There should be overall ten to twenty partitions per shard and number of partitions cannot be changed once deployed. Because of its data partitioning, Oracle NoSQL database is highly scalable as it provides dynamic data partitioning and distribution. Because of no single point failure is it highly available.

The values are created using AVRO Schema which is used to define the data schema for record's value. This is created using JSON format. The following example shows the AVRO schema with JSON format.

```
{
   "type": "record",
   "name": "EmployeeInfo",
   "namespace": "avro",
   "fields": [
      {"name": "name", "type": {
         "type": "record",
         "name": "FullName",
         "fields": [
            {"name": "first", "type": "string"},
            {"name": "last", "type": "string"}
         ]
      },"default":{}},
      {"name": "id", "type": "int"},
      {"name": "address", "type": {
         "type": "record",
         "name": "Address",
         "fields": [
                  {"name": "street", "type": "string"},
            {"name": "city", "type": "string"},
            {"name": "state", "type": "string"},
            {"name": "zip", "type": "int"}
         ]
      },"default:"{}}
   ]
}
```

Updating and creating the record are identical operations. A key needs to be constructed by specifying its major and minor key components. The following is an example for creating and putting values.

```
Key myKey = Key.createKey(majorComponents, minorComponents);

Value myValue = Value.createValue(data.getBytes())

Kvstore.put(myKey,myValue);
```

For deleting records, kvstore.delete() method is used to delete a single record and kvstore.multiDelete() to delete multiple records. The following is an example:

For reading records kvstore.get is used. This returns a value version object and valueversion.getValue() to retrieve the value object of the associated key

Indexing

The data is retrieved using the primary key, but when you create indexes you don't have to completely depend on primary key. Indexing creates dissimilar primary key values but with some shared characteristics. For example, if you want to retrieve an item you might retrieve a barcode which is a primary key. But if you want to retrieve all items used for some purpose, creating an index on a field is best way. In Oracle NoSQL, creation of indexes takes a lot of time because it must scan all the data contained in the relevant table. There are indexing arrays, indexing maps and indexing embedded records

Oracle NoSQL supports table API also. We can use kvstore.getTableAPI() method to get the Table API interface instance and to create a handle for the table use. To create a Table API the syntax is:

TableAPI table = kvstore.getTableAPI();

Table student=table.getTable("Student).

To create a row instance:

Row row = student.createRow();

And then to populate the field we need to use put method.

To delete we use TableAPI.delete(). We use table.get() method to read. It is almost similar to Key-value API, but table API is mostly used when you require ACID transactions.

Transaction management

Oracle NoSQL as any other NoSQL database is not very inclined towards ACID properties. There is a clause to this. Atomicity is supported in few cases where an atomic operation is performed on the key similar to both major and minor components. It is called as practically ACID.

Query management

The query management is Oracle NoSQL is efficient because of the major and minor key components present in the data storage. Oracle NoSQL Database provides key access methods including multi-key variations. The database can also be accessed using SQL as an external table from within a relational database. The key comprises of major and minor components. Keys are spread using hash across partitions based on major components. Every key must have at least one major key component. Records that share the same combination of major key components are to be in the same partition. When the records are in the same partition then querying is easy. For example, /alankritha/yata are major components of the key. In /alankritha/yata/-/birthdate the birthdate is the minor component. The value is the data that you want to store, retrieve and manage. The external tables also can be queried using Oracle NoSQL as it supports Table API.

2.1.2 Simple DB

Amazon's SimpleDB is a key-value store which became famous because of its ease of use. The data in this looks something similar to tables and items within each domain. Each item in here has key-value pairs. It is a distributed database which is written in Erlang. It provides core

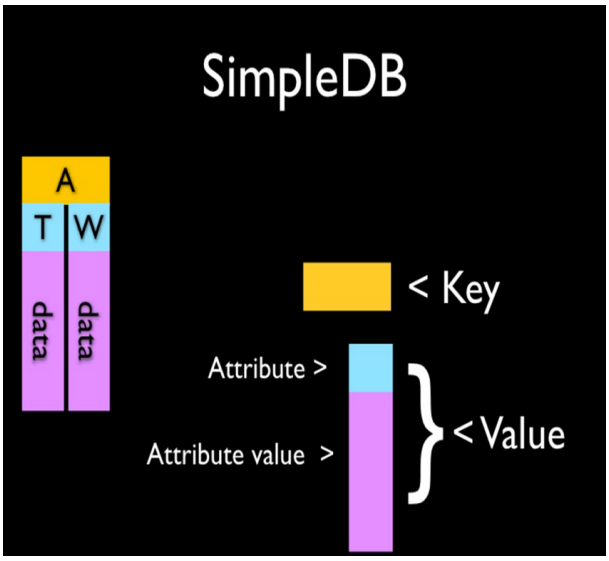
functionality of a RDBMS such as real-time lookup and simple querying of structured data. Unlike other databases, it doesn't require a schema itself. It automatically indexes the data and provides a simple API for storage so that there is no administrative burden.

Feature.

Simple DB's features include flexibility, easy to use, fast, reliable, scalable and inexpensive. Because this DB doesn't require a schema, it doesn't require pre-defined functionalities stating its flexible nature. The fully indexed data of this database is stored redundantly across multiple servers which ensures it reliability. Amazon SimpleDB provides streamlined access to the lookup and query functions. This is something similar to RDBMS and this promotes its ease to use characteristic. Because of its quick, efficient storage and retrieval of data we can conclude saying that it is fast.

Data management.

This DB deals with structured data and this data is organized in the form of domains in which you can put, get, delete or run queries. We can call domains something similar to tables. These domains consist of items similar to attributes. the main components of a data storage in SimpleDB are domains, attributes, items, and values. Regardless of how we store data, this database indexes the data for retrieval purposes.



The above example clearly explains the simplicity of SimpleDB where "A" is a key and "T", "W" are attributes with values. The attribute and attribute value comprise of values. The following given an example how to put attributes into the database. Example: PutAttributes.new(:domain =>'test', :name=>'EmpName', :attributes={:name=>'yata'}

Similar to above example GetAttributes is other functionality.

Transaction management.

This database satisfies any kind of complex transactions and relations in order to provide unique functionality. It doesn't support joins. There are few transactional semantics which this DB offers such as conditional puts and deletes. These help from preventing lost updates when different resources write concurrently at same time.

Query management.

The querying in SimpleDB is very similar to RDBMS. Though we cannot perform joins in simpleDB just like other NoSQL database, it supports counts, sorts and range queries. The queries can be tuned

using composite attributes. The composite attributes are possible because of the key-value pairs.

2.1.3 Aerospike

Aerospike is a real-time distributed database platform which focusses on maintaining high performance and scalability without compromising the traditional database fundamentals such as consistency and reliability. Explosion in the internet applications like location-based applications, advertising etc. are well known. Major internet companies like Facebook, Amazon are facing high write loads and it is reasonable for them to have such issues. But it is observed that even small and medium sized companies are facing the same problems. The NoSQL databases like Mongo DB, VoltDB etc. are supposedly providing solutions to these problems, but their lack of providing consistency and reliability is forcing a lot of companies to choose Aerospike as their database. The system architecture and key technology behind the system will give us a good insight about Aerospike.

Features.

The features of Aerospike include a flexible data model, user defined functions, aggregations, queries, geographic replication and a key-value store. It supports a structured data model which is schemaless. Cross language compatibility is the aspect which lets the database have data with different languages. The user defined functions of this database helps in decreasing the network traffic. It is a simple, personalized and instant database. In key-value store of Aerospike, users can perform an atomic operation and set TTL (time to live) on a record. The replication system reflects any changes among all nodes automatically to survive any disaster management.

Data management.

This database supports enhances key-value operations. It provides “CAS”(safe read / modify / write) operations. Data is structured into bins and each of this bins stores different kinds of values. The data management includes data replication for high availability, SSD optimization, cross datacenter replication. It supports complex data types like lists and maps. The data doesn’t conform to a rigid schema. Data is collected to containers called as ‘namespaces’ where data is divided to sets and records similar to tables and rows in RDBMS respectively . Each record is composed of key, metadata and bins. Namespace is similar to a table and it contains records. These records can belong to a logical container called sets. It has the ability to group records.

Indexing.

As this is a key-value store, indexing in this provides the most predictable and fastest access to row information. There are two kinds of indexes in Aerospike. They are primary index and secondary index. The fastest and most predictable index is primary key index. The secondary indexes have the ability to model one to many relationships. These indexes are specified by bin by bin basis. We can create and remove indexes according to our own convenience. The primary key index is a combination of distributed hash technology with a distributed tree structure for each server. They key spaces are partitions using this functionality. Each index consists of write generation, void time and storage address. Secondary indexes on the other hand are stored in RAM for fast look-up. They contain pointers to both master records and replicated records in a cluster.

Transaction management.

Aerospike implements application-level transactions around multiple changes to a single record. Each record in here can contain multiple bins and one or more secondary indexes. Aerospike's architecture is called as "Shared Nothing" architecture because there are no managers or masters and there are no slaves. Having Master/ slave leads to single point failure and inefficient resource usage which is avoided in this database. There is a strict separation at network level between the client and server similar to relational databases. The architecture is divided into two major parts which are database cluster architecture and client layer. Every node in the database architecture consists of distribution layer and this consists of three modules. Transaction processing is one of the module. The transaction processing module handles reads and writes. It also makes sure that any changes happened to one copy of data is coordinated to the other copies.

Query management.

This database allows you to tune various parameters for queries across the cluster. The factors in query management include the number of threads allocated to queries and the ability to efficiently process small data sets. It provides a value based look-up through the use of secondary indexes. As mentioned earlier, an important feature of Aerospike is user defined functions, this helps in simplifying queries.

2.1.4 Riak

Riak, a distributed database where data is spread among several servers so that maximum availability can be provided. This can be either eventually consistent or strongly consistent and it depends on us to choose according to our application. It can even mix and max this feature in its applications. The main goal of Riak is availability, scalability, master-less and simplicity. It offers writes and reads on multiple servers which provides availability, it distributes data around the cluster, so that no matter how much data you add it doesn't get overloaded ensuring scalability, it doesn't support master-slave architecture and machines can be easily added which ensures simplicity. It is a dynamo inspired key-value store written in Erlang with C and C++. It has remarkably high uptime and grows with you. The sweet spot of Riak is high variety of information, high velocity and high volume.

Features.

The best feature of Riak is its availability. Because it data is spread throughout many servers the system is always available. It has low latency and high scalability. Because of data being spread among clusters, the performance is high and the response is fast providing scalability. Any number of machines can be added to increase scalability. It supports consistent hashing which helps data to be distributed evenly among the servers. It even supports intelligent replication which allows you to read, write and update data even after the node goes down. It's hinted handoff feature helps to handle node failure.

Data management.

This database stores keys against values. Keys are grouped into higher level namespace called as buckets. These buckets are used to define virtual key space for storing objects of Riak. This doesn't have any defined data types and can stores values of any data type. There are key-value pairs inside the bucket. It supports two API's namely HTTP and protocol buffers. Even this support LOB that is large object values.

The data types in Riak are called convergent replicated data types.

To create a node, no node should be running. So we can use “riak stop” to stop the node. “riak start” is use to start the new node. If we consider keys, they are binary values used to identify objects. Objects are the only unit of data in Riak’s data storage

Indexing.

There are numerous ways to index document in Riak. The easiest to index document is <INDEX> <PATH>. Riak search indexing must be enable of key-value pairs of data. Secondary indexes in Riak has the ability to tag an object stored in Riak with one or more values. The user must exactly tell what attribute to index on as the data is opaque. This allows two types of secondary attributes: integers and string. The secondary indexes are used when we need to search for data based on terms rather than objects. These indexes use document based partitioning. They are similar to HTTP headers. During write time, the objects are tagged with index entries consisting of key/value metadata. Indexes are spread throughout the machines. The indexes can be modified by reading or adding an object.

Transaction management.

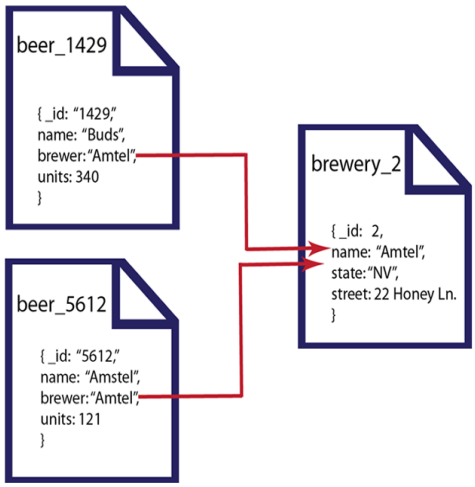
Riak doesn’t support transaction management; especially ACID transactions. Providing transaction support would kill the most important aspect of Riak which is availability. This follows BASE acronym and agrees about the fact that distribution of data is never perfect and the data is eventually consistent in here.

Query management.

The querying in Riak is done in four ways. The general get/put/delete, MapReduce, full-text search, and secondary indexes. Secondary indexes in Riak lowers the cost of finding non-key values. It is done in two ways: exact match or range match. MapReduce is a method of processing data in two phases - map and reduce. In this process it aggregates the data. By performing this, querying can be efficient on the opaque data of Riak.

2.2 Document-based Store

Document-based stores are very similar to key-value stores with the difference of storing in documents rather than key-value pairs. The documents are complex compares to key-value pairs. The documents are a collection of other key-value documents and they are stored in a format similar to the JSON format. Simply put, these data stores are the next level of key-value stores. Here the keys can have the nested values. Even these are schema-less, hence adding data will not be a strenuous task. The documents are self-describing and hierarchical in structure. [3] They are indexed using B-tree and can have nested documents.



In the above figure, we can understand that beer_1429 is a document who has a brewer “Amtel“ and “Amtel” has a nested document about the brewery. One key [2] difference between document store and a key-value

store is in the way document store embeds attributed data associated with stored document that helps in querying. This helps in rapid application development as we can query the data and store a large amount of data.

If we try to reflect back with a CAP theorem, Document store database adapt availability and partition tolerance little similar to key-value store. Though we believe that it can store a large amount of data, it gets confusing after a certain point as the data gets messy. The nested documents and collection of documents make the retrieval complex. Mongo DB, Couch DB, and Amazon Dynamo DB use document store as their data model.

2.2.1 MongoDB

MongoDB is a highly available, scalable and a flexible database. Though NoSQL databases are showing their importance, the RDBMS are not losing their importance. But databases like MongoDB and Aerospike are trying to implement traditional database features and NoSQL features which are making them stand out. For example, though MongoDB is flexible because of its document store database and highly available because of it master/slave architecture. It gained importance because of its RDBMS features like sorting, range queries and secondary indexes. In the recent Facebook session held at RIT, they have explained how they moved to MongoDB. They clearly mentioned that MongoDB is not magic. It did not come out of the blue. It became so famous because of its hybrid nature. It implemented it features basing on all databases be it SQL or NoSQL. For example, it's MapReduce framework from Big Table, consistency from Dynamo. But we need to understand that, it has its own flaws also. MongoDB doesn't support join's. We need to join the data manually without code which is tedious. Other disadvantages of this are memory usage and Concurrency Issues. Because it has to store key name within each document, it takes up more space. I have already written a paper discussing the master/slave nature of MongoDB and sharding of it. Now let us discuss the installation and a basic application performing the CRUD operations.

Features.

It is a schema-less database which supports aggregation tools and map reduce functions. The best aspect of MongoDB is its support to secondary index and geo-spatial index. Because of it's replication process and sharding it is very easy to administrate this database in case of failures. This database is designed to provide high performance and store humongous amount of data without any failures.

Data management.

Mongo DB stores all the data in the form of BSON format which is collection of keys and values. And these keys and values are stored in the form of documents. The maximum document size is 16MB. All documents are stored in records and all the records together form collections. Every collection has set of documents which are logically related.

A sample Mongo DB document:

{ name: "Alankritha", age: 26, Hobbies: ["swimming" , "Reading"], Course: "Computers"}

The documents in the same collection usually share common fields, structure and indexes. Every document stored in a collection has a unique id which acts like primary key. If we don't specify the

id Mongo DB uses ObjectId as a default value. We can have document inside documents called as nested documents and this helps us to retrieve data easily as Mongo DB doesn't support joins. This format of storage helps decrease the I/O cost.

Indexing.

The efficient execution of queries in Mongo is because of indexes. If it's not for indexes, it would have to scan the whole collection to retrieve data. For example, if there is a query,

db.Craiglist_users.find({age:25})

The above pictorial representation shows how indexes decreases the number of documents to be scanned to retrieve the required result for the query. The index structure keeps filtering its process simultaneously with the query optimizer. This database has a very well explained indexing system. We can create indexes for documents and records

Transaction management.

Though management of databases is very easy, transaction support is very less. If your application is focused on transaction management then this DB is not what you have to use. It doesn't implement Multi-Object transactions. It doesn't provide atomic transactions on documents. But a very interesting feature of MongoDB is it allows users to view inserted documents and modify them before it commits. It follows a two phase commit protocol. The phase 1 is to retrieve the transaction to start then it updates the transaction state to pending. Apply the transactions to all the documents in use. Then updating the transaction state to applied then to done.

Query management.

MongoDB has a set of query operation which are very user-friendly. For example show collections show all the collection in the database, find() retrieves requested documents in a particular collection specified. There are a lot of inbuilt functions in MongoDB to insert, remove and find data. Let us consider few queries,

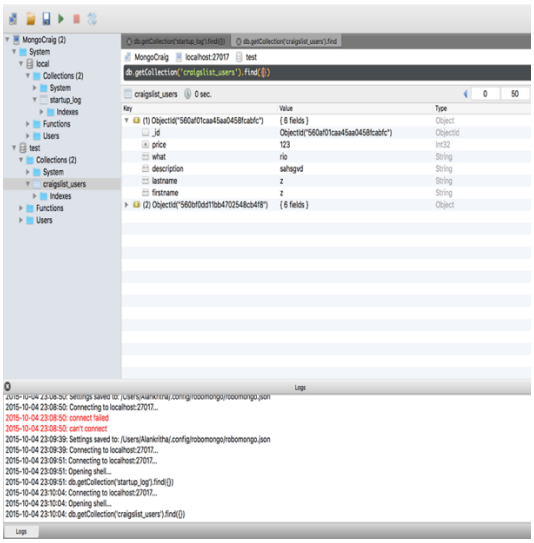
1. db.craigslist_users.find() retrieves all the documents in the collection list.

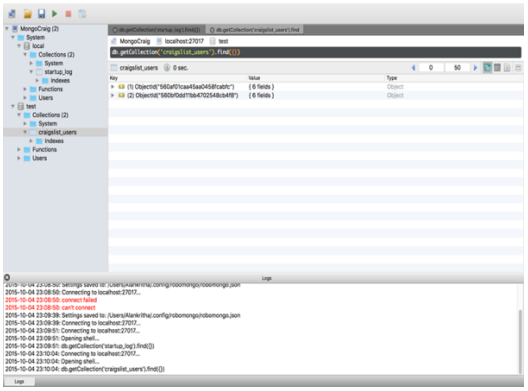
2. db.craigslist_users.find({age:26},{_id:0}) retrieves documents whose age value is 26.

The format of the query is very important because if we miss even one parenthesis, we cannot process the query. MongoDB uses query plans for query optimization. Whenever the query runs, query system uses query plan to chose the most efficient query plan. The query optimizer runs the query against all the plans in parallel. It checks if there are any common records that are found in the buffer. These usually contain ordered, unordered and either of the plans. It also checks if the results or query plan that is already executed is already executing in the queue. If there is one which is already executing, then it skips duplicates. If there are no duplicates, testing stops and optimizer selects other index.

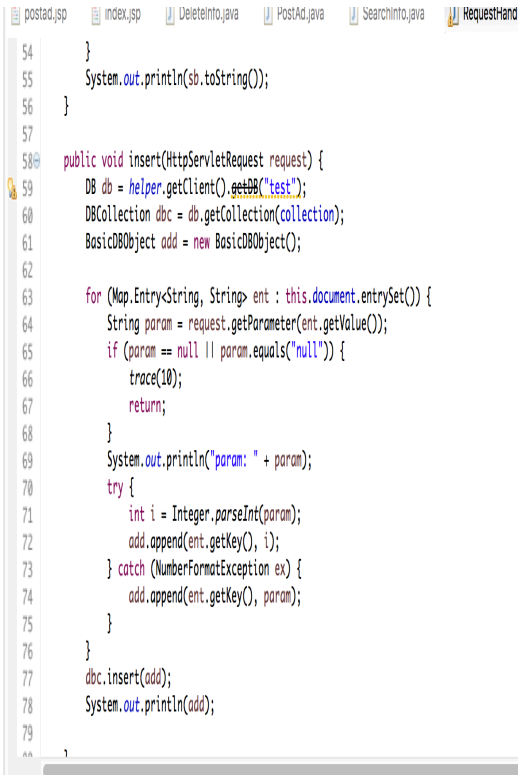
Implementation.

Installation





Starting the Server.



```
}
System.out.println(sb.toString());
}

public void insert(HttpServletRequest request) {
    DB db = helper.getClient().getDB("test");
    DBCollection dbc = db.getCollection(collection);
    BasicDBObject add = new BasicDBObject();

    for (Map.Entry<String, String> ent : this.document.entrySet()) {
        String param = request.getParameter(ent.getValue());
        if (param == null || param.equals("null")) {
            trace(10);
            return;
        }
        System.out.println("param: " + param);
        try {
            int i = Integer.parseInt(param);
            add.append(ent.getKey(), i);
        } catch (NumberFormatException ex) {
            add.append(ent.getKey(), param);
        }
    }
    dbc.insert(add);
    System.out.println(add);

}
```

```
public void retrieve(HttpServletRequest request,
        HttpServletResponse response) {
    DB db = helper.getClient().getDB("test");
    DBCollection dbc = db.getCollection(collection);
    BasicDBObject add = new BasicDBObject();

    try (PrintWriter out = response.getWriter()) {
        /* TODO output your page here. You may use following sample code. */
        out.println("<!DOCTYPE html>");
        out.println("<html>");
        out.println("<head><script>"
                + "function myFunction(elt) {"
                + " elt.setAttribute('selected_object_id', elt.getAttribute('id'));"
                + " console.log(\"attr: \" + elt.getAttribute('id'));"
                + " console.log(\"attr2: \" + elt.getAttribute('selected_object_id'));"
                + "var xhr = new XMLHttpRequest();"
                + "xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');"
                + "xhr.onload = function () {"
                + "   console.log(this.responseText);"
                + "};"
                + "    xhr.open('POST', 'http://localhost:8080/MongoCraig/DeleteInfo', true);"
                + "    xhr.send(\"selected_object_id=\" + elt.getAttribute('id'));"
                + "}" + "</script></head>");
        DBCursor cursor = dbc.find();

        out.println("<body>");
```

```
public void update(HttpServletRequest request, HttpServletResponse response) throws IOException {
    //two possibilities:
    String par=request.getParameter(DataKey.ADVERT_ID.getKey());
    if (request.getParameterMap().size() > 1) {
        //the parameters have other values we need to assign into the db
        //use #getParameter with the value from the #getKey on the form
    } else {
        //ONLY the advert id exists in the parameters
        PrintWriter out = response.getWriter();
        Map<String,Object> updating=helper.getInfo(par);
        out.println("<html><body>");
        out.println("<form method=\"POST\" action=\"" + Page.EDIT.getPage() + "\">");
        //iterate over map, make input element for each value
        out.println("<input type=\"text\" name=\"" + DataKey.ADVERT_ID.getKey() + "\" value=\"" + par + "\" style=\"display:none;"
        for (Map.Entry<String, Object> entry : updating.entrySet()){
            out.println(entry.getKey() + " <input type=\"text\" value=\"" + entry.getValue() + "\" name=\"" + entry.getKey() + "\"
        }

        out.println("<button type=\"submit\">" + "UPDATE" + "</button>");
        out.println("</form>");
        out.println("</html></body>");
    }
`
```

2.2.2 CouchDB

CouchDB was developed by Apache software foundation and it is an open source database. It is written in Erlang language to support the use of unstructured or semi-structured data. The focus of this database is on the ease of use and embracing the web. The database strength is the way developers can create and deploy web apps. In CouchDB, we don't need the server anymore as the database itself acts as a server. This makes it very simple for developers to set up a web application. CouchDB can be operated using two different methods. The first one is the command line utility called curl. The second way to access the DB is using the built-in administration interface called the Futon. The best aspect of CouchDB is it features ACID (Atomic Consistent Isolated Durable) properties. As it doesn't overwrite the committed data on the disk, the data is always consistent. This database uses MVCC (Multi-Version concurrency control) model and the documents are indexed using B-trees. When data is updated in documents, the database is always in a consistent state.

Features.

The main features of CouchDB are storage, ACID Semantics, Map/Reduce Indexes, distributed architecture with replication, transaction Support, REST API and Security and Validation

Data management.

CouchDB's database is a document store data model. The data in here is stored in records and it is a self-governing data file. Records are the primary components of data storage. Each record can have any number of fields in them. It is schema-less. Therefore, addition of data anytime is possible without any problem. As already mentioned it supports ACID properties. So to support atomicity, the documents edited are either saved completely or not saved at all. The documents have different names in the database and these documents are CouchDB's central data structure. This data model is built around saving documents as users can come back any time and resume their work. In other database there are timestamps to do this work, but in here the entire control is given to the user. Because of the presence of validation functions [] users don't have to worry about the bad data. The format of documents are as follows:

{

"_id"="Hello",

"_rev"="124632675",

title="Hello",

"created_at"="05/12/2015"

}

```
{
   "_id": "00a271787f89c0ef2e10e88a0c0001f4",
   "_rev": "1-2628a75ac8c3abfffc8f6e30c9949fd6",
   "item": "apple",
   "prices": {
       "Fresh Mart": 1.59,
       "Price Max": 5.99,
       "Apples Express": 0.79
   }
}
```

In the above example _id can be chosen either by the client or server and _rev is used for concurrency control. This acts as a gatekeeper to control the writes of the document. Views are used in this database to query and update data. These build indexes that are used to locate documents by values and draw relationships. These is another aspect of data model that is design documents. When permanent views are created they are stored in these special documents called as design documents. The design documents contain the application code and they run inside the database.

Indexing.

As already mentioned CouchDB uses B+ trees for indexing the data. B+ trees are used for indexing by most of the databases as they don't have[3] data associated with interior nodes. Because of this characteristic more keys can fit on a page of memory. In this case there will be few misses in the process of storing and trying to retrieve data. Since the height of the tree is manageable, there is fast access in B+ trees. CouchDB implemented indexing through MVCC design and an append-only design. The index of views is one B-tree. This index helps in fast querying. These documents also contain data with JSON format. Temporary views and permanent views are two kinds of views that contains indexes with B+trees.

Transaction management.

Many NoSQL databases don't support transactions as it is difficult to manage them while keeping up the performance. Couch DB supports single document transactions. This means that it either completely writes the document and then saves it or it doesn’t do anything. When we try to update document it checks with the database’s version and time it modified. If it is different the changes made doesn’t appear.

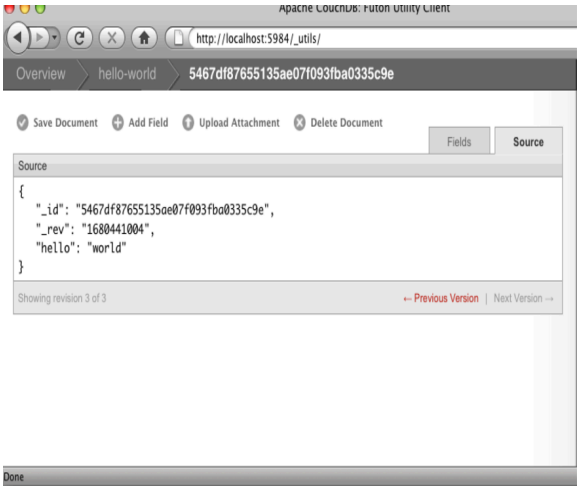
Query management.

In traditional RDBMS, since the data is structured querying was easy, but in databases like CouchDB Map Reduce functions are used for querying. These functions provide flexibility because they can adapt to various changes in the document structure. Added functionality is that indexes can be computed independently and in parallel. The combination of map and reduce is called view in the CouchDB. These views are stored as rows and are kept sorted by key.

Implementation.

Installation

Loading and retrieving data on Futon client



2.2.3 Amazon Dynamo DB

Dynamo DB[9] is best at handling high level traffic. The scaling up and scaling down of tables throughput without any performance degradation and throughput is the aspect with keeps Amazon Dynamo DB apart. This database addresses issues like scalability, management and reliability. In the following section we will understand more about Amazon Dynamo DB by discussing its architecture, data management, transaction management and how it is better than RDBMS and other NOSQL databases. Amazon Dynamo DB is a fully managed database service and it is accessible via simple web services. The architecture of the storage system of Amazon Dynamo DB for production is very complicated. The system needs to have robust and scalable solutions for load balancing membership and failure detection. [1]

Dynamo DB uses port 8000 by default and the command line:

Java-Djava.library.path=./DynamoDBLocallib –jar DynamoDBLocal.jar –sharedDB is used to run dynamoDB locally.

Features.

The best aspect of Amazon Dynamo DB is that it can adapt to both key-value store and document-store database. Because everything here is managed by cloud it has seamless scaling capability. In the presence of autonomous and synchronous replication, it acquires high availability. It is schema-less and can be developed with ease because of the AWS console.

Data management

Data items in this database are stored in solid state disks and are spread across to maintain reliability. The data is stored in the form of key-value pair. The data is stored in tables. Each table contains items which is uniquely identifiable by other items and all the items contain attributes. An attribute is a fundamental data element. This is not broken any further. The following picture shows sample to create table. As you can see, while creating a table you mention if it's a partition key or not. sAmazon web services provide this interesting interface

For example: Department is an item ; id, names, employees etc. are attributes of that item. As usual every table has a primary key and secondary indexes. There are two kinds of primary keys. They are:

1. Partition key: Simple primary key composed of one attribute

2. Partition key and sort key: Composite primary key consisting of two attributes.

Dynamo DB support list and map data types.

The following is the example:

{

Day: "Tuesday",

UnreadEmails:12,

ItemsOnDesk: [

“Coffee Cup”,

{

Pens={Quantity:4},

} }]

Indexing.

Secondary indexes are data structures that contain subset of items from table. There are two types of secondary indexes namely local secondary index and global secondary index. Local index is scoped to a table partition with the same hash key. The secondary index maintains a range key for a given hash key. Once a table is created, the attributes in the secondary index must be specified. All the data in the secondary index is organized on the hash key, with different range key.

In the case of global secondary index parameter of the create table, an operation is used to create one or more global secondary indexes on a table. The name global is derived from the fact that its queries can span across all the data in the table and across all partitions with different hash keys.

Transaction management.

When multiple users are accessing a table between reads and writes there is a possibility that the updates made by one user might be overwritten by other or changes made by one user might not be seen to another user. To solve these issues Dynamo DB supports few features, namely, Conditional write, Atomic counter read consistency and fault tolerance.

In a conditional write, a condition is specified by the user while trying to update an item. The database writes the item only when a condition is satisfied. For example, the user can specify a value and update an item. If the value is same on both client and server side only then update is written otherwise it return an error.

DynamoDB also uses atomic counter. Update item keyword is used in this scenario without interfering in other requests. DynamoDB maintains multiple copies of each item to ensure durability. Therefore, when a write request is successful, [4] it should make that change to all servers. Fault tolerance is a built in feature of DynamoDB. This feature synchronizes and replicates data across multiple availability zones in a region. This is to ensure the availability of data and to avoid failures.

Query management.

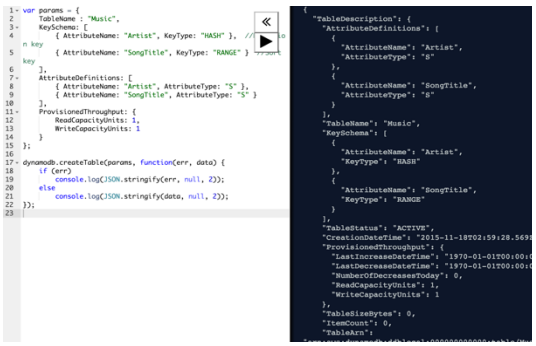
Query operation works by using primary key attribute values. The requirement for searching is to use hash key attribute name which is distinct value. Range key is optional for a search operation. Using Java, the CRUD operations can be performed. In the basic CRUD operations, you will see examples.

We mentioned that id is a primary key and it holds numbers. DynamoDB uses JSON format to send and receive formatted data. It is human readable content and consists of attribute value pages. To query a table, we can put conditions only on the primary key attribute of the table. For equality condition search, a hash key attribute is used.

Implementation

```
Alankrithas-MacBook-Pro:AmazonDynamo Alankritha$ java -Djava.library.path=
moDBLocal_lib -jar DynamoDBLocal.jar -sharedDb
Error: Unable to access jarfile DynamoDBLocal.jar
Alankrithas-MacBook-Pro:AmazonDynamo Alankritha$ cd dynamodb_local_2015-07-
0
Alankrithas-MacBook-Pro:dynamodb_local_2015-07-16_1.0 Alankritha$ ls
DynamoDBLocal.jar       LICENSE.txt             third_party_licenses
DynamoDBLocal_lib       README.txt
Alankrithas-MacBook-Pro:dynamodb_local_2015-07-16_1.0 Alankritha$ java -Dj
brary.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -sharedDb
Initializing DynamoDB Local with the following configuration:
Port:   8000
InMemory:       false
DbPath: null
SharedDb:       true
shouldDelayTransientStatuses:   false
CorsParams:     *
```

Inserting values



Querying

Creating

2.3 Column-based Store

The idea behind [6]Column-based is to store humongous amount of data. The data here is stored in cells which are grouped in columns of data rather than rows of data. This might look similar to tables, but it isn't as this is schema-less and has keys pointing to multiple columns and those columns arranged accordingly. This data model comprises of key, column, column family and key spaces where keys have different number of columns, columns have list of values, column family has a group of columns and key spaces are like the names of the table by which you can recognize it. The read and write function are done using columns. In this model all cells corresponding to a column are stored together which makes searching and querying a piece of cake.

Row ID	Columns...		
1	Name	Website	
	Preston	www.example.com	
2	Name	Website	
	Julia	www.example.com	
3	Name	Email	Website
	Alice	example@example.com	www.example.com

In the above example if you see the columns family, first and the second row contains two columns whereas the third row contains three columns. The rows can have the different arrangement of the columns and this is the best aspect of this data model. If the website columns have sub-columns such as domain, namespace then the website will be considered as a super column.

If we try to understand this using CAP theorem, column-store data models support consistency more often and incline a little towards partition tolerance. High scalability, indexing, and sorting are the

advantages of this data model. Though this data model is considered to be the best amongst other models, handling transactions with this data model is significantly slower. If you are willing to develop an application with a good transactional support, implementing this data model for that application would be a bad choice. HBase, Cassandra, and Big Table belong to column-store databases.

2.3.1 HBase

HBase is a distributed database built on the top of HDFS (Hadoop distributed filesystem). It is built on java and has a column store database. It is started as a project by Powerset and is modeled by Google's BigTable. This is a highly scalable database and sharding is done on the rows easily for load balancing. This has a high read and write throughput, and is the best database to use when dealing with humongous amounts of data.

Features.

The key feature of HBase is it is not eventually consistent. It is strongly consistent which makes this suitable for all tasks including transactions and aggregations. It has automatic sharding, meaning the tables are distributed in the cluster and are split and redistributed according to the requirements. It supports an easy to use Java API and is built to HDFS. It provides built-in-webpages. Versioning and Compression are two more inbuilt features wherein the versioning timestamp is maintained for each value of the column and in compression similar values are grouped together. It supports Bloom filters for high query optimization.

Data management.

It is a column-oriented database where columns are grouped into column families. In the below figure we can clearly understand the data management. The row key identifies each row. It is like the primary key or unique key. [10[Below Customer and Sales both are two different column families. Clearly states that there can be multiple column families for each row. There are time stamps related to the data so that the users can resume their work anytime. The column family has multiple column qualifiers and these qualifiers have timestamps for each version.

Row Key	Customer		Sales	
Customer Id	Name	City	Product	Amount
101	John White	Los Angeles, CA	Chairs	$400.00
102	Jane Brown	Atlanta, GA	Lamps	$200.00
103	Bill Green	Pittsburgh, PA	Desk	$500.00
104	Jack Black	St. Louis, MO	Bed	$1600.00

Indexing.

Similar to other databases even HBase uses B+tree because it provides high performance. The RowKey is used as a primary key index in the indexing by default. This supports random read and write. Because of the Row-key and table like structure HBase can locate any region in the database. There are steps involved to detect the data. A client locates the region server that hosts -ROOT- region. Here the ROOT is a catalog table that keeps track of the location of all META tables. Then client tries to find-META- region through ROOT region. Here META is also a catalog table that maintains the list of all the region servers containing the actual data. After this the final region is found.

Transaction management.

If we talk about a transaction, the atomic operations can be performed on tables that are cell values. Because of MVCC (Multi-version concurrency control) time stamps are produced which act as transaction Id's. All the writes and reads are performed using these transaction ID's. This database support HBase in limited ways. It doesn't support mixed read/write transactions. The transactions are committed serially in this database.

Query management.

There is a bloom filter in the end. Bloom filter is a data structure that determines if an element exists or no. If the Bloom Filter returns true for a block, the Block indexes to find the block that has the RowKey. This is similar to B+ tree index. We can retrieve the required data using this RowKeys. Creation of tables helps in efficient querying.

2.3.2 Cassandra

The solution to store humongous amount of data without compromising performance, consistency, availability and partition tolerance. Cassandra is one such NoSQL database which improved rapidly. It was introduced in the year 2012 by few engineers of Facebook who combined the functionalities of Amazon DB and Big Table. This has a key Value API.

Data management.

Cassandra gets its data model from Dynamo white paper by Amazon and its data representation from the BigTable whitepaper by Google. Tables are created by CQL(Cassandra Query Language) and this is where the data is stored and distributed. Similar to RDBMS even these tables have PRIMARY KEY and this is also called as a partition key because this key distributes data. But this doesn't mean Cassandra [11]data model is anything like relational data model. Relational models are designed for efficient storage whereas Cassandra is designed for storing large amounts of data and for performance. For example, we have to store dog, cat, duck, lion in a table. Then the query would be:

Before creating a column family, a column space should be created. Cassandra stores the column name with each data item. It has a very flexible schema. The column is the basic building block in Cassandra. It is constructed in three parts. They are column name, value, and timestamp. The below picture is an example of a column family. It is an ordered collection of rows, each of which has columns inside. The primary key is the unique identifier. In the below case it's the row_key.

Indexing.

A table is a map indexed by key and the value is an object which is highly structured. Every operation is a unique operation no matter how many columns are being read or written. This has two kinds of column families which are simple and superfamilies. A simple is just one column family and super column family is a column family inside a column family. Depending upon the business the columns can either be sorted by time or data. The columns in a simple column family are accessed by "column_family: column" and that of super are accessed using "column_family: super_column: column". Cassandra API consists of the following three simple methods - insert, get and delete. The syntax is: insert (table, key, rowMutation); get(table, key, columnName) delete(table, key, columName). Cassandra Query language is the interface into the Cassandra DBMS. This is quite similar to SQL, but the

main difference is that CQL doesn't support joins and subqueries. Compaction means freeing up space by merging large data files. During this stage, the data is merged, indexed, sorted and stored in the SSTable. This provides efficient indexing in Cassandra. This also has secondary indexes and this is an efficient way of accessing records without considering primary keys. They are stored in JSON format.

Transaction management.

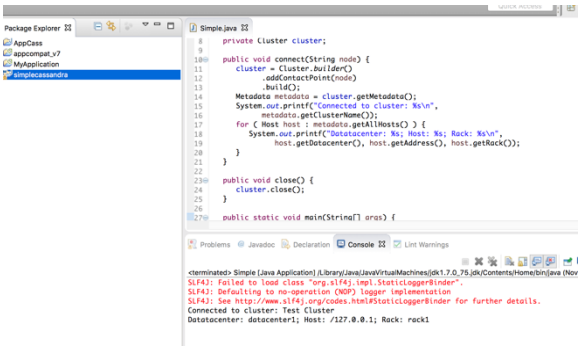
Cassandra writes data to multiple nodes, it either writes all data to the nodes or rolls back as it uses timestamps to resume work. It follows BASE acronym by supporting eventual consistency. Multiple transactions occurring at the same time should not have an impact on each others' execution. Recent updates of Cassandra support row level Isolation. As discussed, SSTables and Memtables ensure that the committed data stays in the database forever to prove the durability of the database.

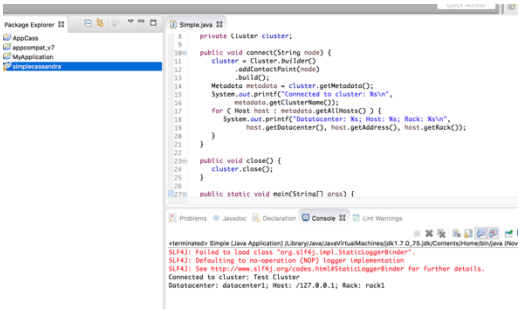
Query management.

It uses Cassandra Query language which is similar to SQL. The only difference is that it doesn't support joins and subqueries. It uses key spaces as schema to perform these queries. It uses Select, Insert, Update, Delete, Truncate etc. This provides better query language that many NoSQL databases. Other than joins the sub-queries are also not supported by Cassandra.

Implementation.

```
Alankrithas-MacBook-Pro:sem4 Alankritha$ ls
FCN             GIT             IS              Web Tech
Alankrithas-MacBook-Pro:sem4 Alankritha$ cd IS/Cassandra
Alankrithas-MacBook-Pro:Cassandra Alankritha$ ls
AQUAS Paper.pdf         Caasandra.pdf           RECODS.docx
AQUAS.docx              Cassandra paper.pdf     RECODS.pdf
AQUAS.pdf               CassandraDB.docx
Caasandra.docx          CassandraImplementation
Alankrithas-MacBook-Pro:Cassandra Alankritha$ cd CassandraImplementation/
Alankrithas-MacBook-Pro:CassandraImplementation Alankritha$ cls
-bash: cls: command not found
Alankrithas-MacBook-Pro:CassandraImplementation Alankritha$ ls
dsc-cassandra-2.1.11
Alankrithas-MacBook-Pro:CassandraImplementation Alankritha$ cd dsc-cassandra-2.1.11/
Alankrithas-MacBook-Pro:dsc-cassandra-2.1.11 Alankritha$ ls
CHANGES.txt     NOTICE.txt      interface       pylib
LICENSE.txt     bin             javadoc         switch_snappy
NEWS.txt        conf            lib             tools
Alankrithas-MacBook-Pro:dsc-cassandra-2.1.11 Alankritha$
```





Installation

```
Alankrithas-MacBook-Pro:bin Alankritha$ ./cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 2.1.11 | CQL spec 3.2.1 | Native protocol v3]
Use HELP for help.
cqlsh> create keyspace demokey with replication={'class':'SimpleStrategy','repli
cation_factor':3};
cqlsh> describe keyspaces;

space  devjava          system   demokeyspace   "devJava"
demo   "devJavaSource"  demokey  system_traces

cqlsh> use demokey;
cqlsh:demokey> create table emp(empid int primary key,empname text,);
cqlsh:demokey> select * from emp;

 empid | empname
```

INSERT

```
cqlsh> use demokey;
cqlsh:demokey> create table emp(empid int primary key,empname text,);
cqlsh:demokey> select * from emp;

 empid | empname
-------+---------

(0 rows)
cqlsh:demokey> insert into emp(empid,empname) values(1,'alankritha');
cqlsh:demokey> insert into emp(empid,empname) values(2,'manogna');
cqlsh:demokey> insert into emp(empid,empname) values(3,'');
cqlsh:demokey> select * from emp;

 empid | empname
-------+------------
     1 | alankritha
     2 |    manogna
     3 |

(3 rows)
```

UPDATE

```
     3 |

(3 rows)
cqlsh:demokey> update emp set empname='hardik' where empid=3;
cqlsh:demokey> select * from emp;

 empid | empname
-------+------------
     1 | alankritha
     2 |    manogna
     3 |     hardik

(3 rows)
```

DELETE:

```
 empid | empname
-------+------------
     1 | alankritha
     2 |    manogna
     3 |     hardik

(3 rows)
cqlsh:demokey> insert into emp(empid,empname) values(4,'sandeep');
cqlsh:demokey> insert into emp(empid,empname) values(5,'Shweta');
cqlsh:demokey> delete empname from emp where empid=1;
cqlsh:demokey> select * from emp;

 empid | empname
-------+---------
     5 |  Shweta
     1 |    null
     2 | manogna
     4 | sandeep
     3 |  hardik

(5 rows)
```

2.3.4 BigTable

Bigtable is introduced by Google in the year 2005. It is just a distributed storage system that is structured as one large table. It is designed to store billions of URL's across several machines. Features of Bigtable are: The data is stored persistently, it stores it data in a Map format where there are keys and values, it is[8] distributed. It also has a time-based feature meaning, every column family has multiple versions of column- family data. If an application doesn't specify timestamp, it will retrieve a latest version of a column family. The tables in Bigtable are sparse means a row can have different columns and those columns can be inconsistent. Bigtable comes under column family. In the following sections we will learn about installation, data management, transaction management and the use cases of Bigtable

Data management.

As mentioned above, big table is sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp. The number of columns in the table can be unlimited and it is not necessary that it should be populated. There is no specific installation for Bigtable as google provides a free trial to work on the cloud platform. We can directly write code either in java, python or ruby on a console and develop applications. Google cloud acts as a database in this scenario. Data model of Bigtable is indexed by row key, column key, and timestamp. We can do a fast lookup using the key and multi-version storage. Since rows are ordered lexicographically, every scan is in order. Bigtable provides functions for creating and deleting column families. It provides various other functions for changing table, cluster etc. The examples of reading from a big table and writing to a big table are as follows:

Writing to and Deleting from a Bigtable:

```
//open table

Table *T= OpenOrDie(/bigtable/webtable);

//writing

RowMutation r1(T,"com.aaa");

r1.Set() ;

RowMutation r2(T,"com.cnn.www");
```

r2.Set(“anchor:www.c-span.org”,”CNN”);

r1.Delete(“anchor:www.cnn.com”);

Operation op;

Apply(&op, &r1);

Reading from a Bigtable:

Scanner scanner(T)

ScanStream *stream;

stream = scanner.FetchColumnFamily(“contents”);

stream ->SetReturnAllVersions();

Scanner.lookup(“<!DOCTYPE html PUBLIC…”);

Bigtable supports the execution of client-supplied[7] scripts in the address spaces of the servers. The scripts are written in Sawzall, a language developed by Google.

Transaction management

Every database strives to provide ACID properties for transaction support. Similarly, Bigtable provides durability and atomicity through the commit log. Google file system helps provide durability as it replicates to three computers, which write them durably onto disks and flash. Regarding atomicity in Bigtable, because of its distributed nature transactions coordinate among tablet servers. The coordination basically would compromise scalability. This simply means that Bigtable doesn’t support arbitrary transactions. This is only of there are multiple tablet servers, but coordination is possible in a single tablet server. Bigtable ensures that data relevant to one table will never be split across two tables. An easy way out is limiting the transactions to support just one key. Though single key transactions are easy to implement, they are too limiting. Therefore, split the key into a column and a row. In this case, splitting keeps all rows together which have keys. RDBMS uses primary and foreign key relationships whereas Bigtable uses maps.

Implementation

Installation

2.4 Graph based

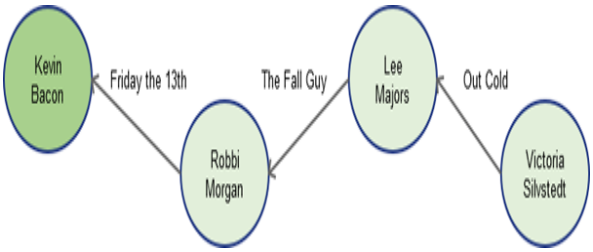
Though graph-based data stored are rare, it gained hype due to few NoSQL databases based on the data store. This structure has edges and nodes. Till now we have noticed the databases in the form of tables, keys and columns but in this we see how each entry is related to other entry something similar to a linked list.[6] As the database grows the relationships between the entry increases leading to complexity.

If you consider the following example, one table has unique id’s and keys and another table has the films associated with the actors. [4]

ID	Actor
1	Kevin Bacon
2	Lee Majors
3	Robbi Morgan
4	Victoria Silvstedt

ID	Actor 1	Actor 2	Film
1	4	2	Out Cold
2	2	3	The Fall Guy
3	3	1	Friday the 13th

Now observe the graph-based data model's relationships, from the table we can say that there is no relationship between Victoria and lee other than id's, but the graph based explains to us that they are related through a film and those two are actors. Graph based helps us find the optimal route as it stores relationships.



Graph based data models are required when data is highly linked to each other and the relationships between them helps us retrieve data. Neo4j and titan are the databases that fall under this data model. Though it has its set of advantages, performance of this data model compared to others is less because of the lack of indexing and querying would be complicated as there exist relationship.

Differences between types of databases based on characteristics of Data Management, Indexing, Transaction Management and Query Processing.

TYPES OF DATABASES	DATA MANAGEMENT	INDEXING	TRANSACTION MANAGEMENT	QUERY PROCESSING
AEROSPIKE	• Schema less • Has namespaces similar to databases which contains sets(tables) and records(rows). • Records have indexed keys and values which are stored in bins.	• Supports primary and secondary index. • Very efficient because of key-value pairs. • Indexes are specified on bins.	• Application level transactions. • It supports atomicity, isolation, consistency and durability.	• User defines functions helps in efficient querying. • Map-reduce also adds to the querying.
ORACLE NOSQL	• Stores data in KVStore in the form of key and value pairs. • AVRO schema is used to define the	• Retrieved using primary key • Creates dissimilar primary key	• Not very inclined towards ACID properties. • Atomicity is supported in selective cases.	• Major and Minor key components have significant role

	data schema. • Also support Table API and LOB API	values & has some shared characteristics • Creation of index is time consuming. • Consists of indexing maps, arrays and embedded records.	• It is characterized by BASE acronym (Basically available, Soft state and Eventually consistent)	• Mostly it discards the transaction functionality and lets the developer handle it.
SIMPLE DB	• Structured data is stores in domains. • Domains consists of items, attributes and values. • Multiple values can be associated for each row.	• B+tree index as it is similar to RDBMS. • Supports secondary indexes.	• Support any complex transactions. • Offers conditional puts and get.	• Similar to RDBMS. • Supports counts, sorts, range queries
RIAK	• Stores data in buckets, keys and values with bucket types. • Buckets are flat namespaces. • Values are identifies by key-value pairs stored in buckets.	• Primary and secondary indexes work. • Secondary indexes have the ability to tag an object stored with one or more values for efficient searches.	• Doesn't support transactions. • Eventually consistent to ensure availability. • Inclined towards BASE acronym.	• Querying is done through secondary indexes, Map Reduce • Since data is opaque efficient is difficult.
CASSANDRA	• Stores in tables (column families) indexed by key and values which are highly structured. • CQL to retrieve data using queries. • Column name, value and time stamp are the attributes.	• Supports secondary indexes and composite keys. • Full text search and geospatial indexes are not supported.	• It supports atomicity, consistency and durability. • It provides no isolation. • It is a lock free model.	• Use CQL for querying. • Similar to SQL but fails to support joins and sub queries.

HBASE	• Stores data in columns which are grouped in column families. • RowKey identifies each row. • Each row is a sorted key/value map. • Multiple column families for each row.	• Uses B+trees for indexing. • RowKey is used as a primary key in indexing. • Supports secondary indexes.	• MVCC helps to maintain ACID properties. • Doesn't support mixed read/writes. • Conditional ACID	• Bloom filters help in querying. • Since tables can be created querying is efficient.
BIGTABLE	• It is a sorted map indexed by row key, column key and timestamp. • Value in a map in un-interpreted array of bytes. • Row key in a table are arbitrary string.	• Supports secondary indexes, full text search.	• Doesn't support atomicity and isolation • Partially supports consistency through commit log.	• BigTable's cloud helps in efficient querying. • Provides Google's Big query.
MONGODB	• Schema-less • Stores in BSON format. • Document is stored in a record and the documents together constitute collections. • Reference and embedded documents.	• Compound indexing & single field indexing • Allows single field indexing. • Supports multi-key indexes, geo spatial indexes and hashed indexes.	• Provides atomic transactions only on a single document. • No multi-document transactions. • Durability, isolation and concurrency are supported with some restrictions. • Uses reader-writer locks for concurrent reads.	• Built-in functions for querying. • Optimal query plan is chosen by running all the plans in an optimizer. • Plans are run on the available indexes. • Complex queries decreases the performance.
COUCHDB	• Stored in JSON format. • Each document contains unique ID and revision number. • B+tree for storing data. • Multiple versions	• Indexes are created using map and reduce functions. • Indexes are stored in the form of views.	• Eventual consistency • Version control based on the document revision number. • No Locking mechanism. • Because of	• REST API is used to query in this database. • Map Reduce functions are used to query the data as they aggregate

	of a document.(MVCC framework)	• Permanent views and temporary views. No full text search.	MVCC supports ACID partially.	the data.
AMAZON DYNAMODB	• In JSON format. • Stored in tables which contains attributes and items. • Two types of primary keys: partition key, partition key & sort key.	• Supports secondary indexes and composite keys. • Full-text search is not supported.	• No Atomicity, but manages to provide consistency, isolation and durability. • Other features are possible because MVCC.	• Primary key attributes values are used for querying. • Hash key attributes are Used to search.

Conclusion

In the course of studying these various NoSQL bases, it is clearly understood that we cannot say that one database is the best. It comprises of advantages and disadvantages and this is the reason why organizations have a combination of NoSQL databases. The combination in this case means, if an organization requires sharding without compromising performance and handling humongous amount of data, then it chooses both MongoDB[5] and Cassandra. MongoDB for sharding and Cassandra for handling performance while dealing with humongous amount of data. Similarly, if it's a financial company who doesn't want to compromise ACID properties and if they are not worried about the data and performance then Aerospike would be the best solution. After analyzing all the databases closely based on indexing, transaction management, query processing and data storage MongoDB and Cassandra emerged to be the best choices based on the above mentioned characteristics. Though Key-value store databases are easy to implement their disadvantages lies while increasing the amount of data. If you need to develop an application within a day or two, then MongoDB should be your first choice. If you don't want to worry about a Schema or waste time in designing a schema, then SimpleDB should be your choice as it doesn't have any Schema. For interactive web application CouchDB should be the choice. Google's Big Table has been a base for designing a lot of new NoSQL databases. After implementing five NoSQL databases, the resources provided for MongoDB were best. It is User interactive. Amazon's cloud services and its online console helps us gain hands-on all its functionalities. HBase and BigTable's query capabilities are helpful for efficient searches.

The list goes on as we try to say which one is best over the other. But before that we need to address that why we need to move to NoSQL. The continuous growth in the research of NoSQL databases and improvising them day by day is making a lot of organizations use them. But still those organizations have RDBMS as their core.

They only implemented few functionalities in their organizations related to NoSQL databases. There needs to more research done in this field to completely replace SQL. Instead combining these both databases that is SQL and NoSQL together gives more productive results. NoSQL cannot be a replacement of SQL. But the increasing data and the need for performance through out the world requires the functionalities of both kinds of databases.

References

[1]http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[2] http://searchdatamanagement.techtarget.com/definition/NoSQL-Not-Only-SQL

[3]http://nosqlguide.com/graph-database/nosql-databases-explained-graph-databases/

[4]http://rebelic.nl/2011/05/28/the-four-categories-of-nosql-databases/

[5]http://www.3pillarglobal.com/insights/exploring-the-different-types-of-nosql-databasesv

[6]http://nosqlguide.com/graph-database/nosql-databases-explained-graph-databases/

[7]http://guide.couchdb.org/draft/documents.html

[8]http://docs.aws.amazon.com/amazondynamodb/latest/gettingstartedguide/GettingStarted.JsShell.05.html

[9http://db-engines.com/en/system/Cassandra%3BHBase%3BMongoDB

[10]http://blog.nahurst.com/visual-guide-to-nosql-systems

[11]http://www.informit.com/articles/article.aspx?p=1619309