ISSN: 2321-7782 (Online)
Impact Factor: 6.047

Volume 4, Issue 8, August 2016

International Journal of Advance Research in Computer Science and Management Studies

Research Article / Survey Paper / Case Study

Available online at: www.ijarcsms.com

NoSQL Database: An Advanced Way to Store, Analyze and Extract Results From Big Data

Kaustav Ghosh[1]
Department of Computer Science
St. Xavier's College, Kolkata
Kolkata – India

Dr. Asoke Nath[2]
Department of Computer Science
St. Xavier's College, Kolkata
Kolkata – India

Abstract: As production of data is increasing by leaps and bounds every second, so are efficient and modern techniques on its way of development and advancement. The data needs to be stored efficiently, kept protected from malicious agents, analyzed properly as well as quickly and results derived as per demands. Storing data in NoSQL databases is such advancement in technology. The article deals with brushing up of the main properties of SQL databases and then going into what kicked off the development of NoSQL databases, the CAP-Theorem. It then clearly explains the key differences between SQL and NoSQL databases and certain characteristics of NoSQL databases, reasons enough as to why we need to use NoSQL databases. NoSQL data modeling techniques are different, though at places similar, to SQL data modeling techniques, hence some of the NoSQL data modeling techniques have been covered. Some of the features and examples of the various types of NoSQL databases are explained to sum it up.

Keywords: Big Data, Join, API, Sharding, MapReduce, LDAP, AD, 32-bit, SSL, CQL, TCP/IP, IDE intelliSense, Refactoring.

I. INTRODUCTION

Traditional database systems are based on the **relational model**. These are popularly called **SQL databases**, named particularly after the **language** that **queries** them. They have been popular for data storage and retrieval for a very long time. However, in the last quite a few years due to **data surge** and a **variety** of them and its **dominance**, **non-relational databases** steeply rose in popularity. These databases are known as **NoSQL databases**. They are quite **different** from traditional SQL databases. Most of these are based on storing that **enhances speed**. With the increased use of **Internet** and the availability of a large number of **handheld devices** and advancement in **technology**, huge amounts of **structured**, **semi-structured**, and **un-structured data** are needed to be captured and stored for a variety of **applications**, **analysis** and **profit**. Processing such vast amount of data, **Big Data**, requires **speed**, **flexible schemas** and **distributed databases**. NoSQL databases proved to satisfy majority of the requirements for operating on Big Data. This led to a **significant increase** in the number of NoSQL database offerings. Today several **commercial** and **open-source implementations** of NoSQL databases like **BigTable** and **HBase**, only to name a few, have come up. Some of them will be covered later in the article.

Relational databases are designed with **reliability** and **consistency** at its cynosure. They focus on the **four principles** of the ACID model that are always preserved in order for **accuracy**, **completeness**, and **data integrity** to be maintained.

A. Acid Principles:

- **Atomicity** – Atomicity is the property which states that a database transaction must be treated as an atomic unit. In other words, **either all of its operations are executed or none at all**. In no situation must there be a state in the database where a transaction is left **partially complete**. States must be defined **either before** the execution of the transaction or **after the execution** (**abortion** or **failure** as the case may be) of the transaction **[1]**.

- **Consistency** – Consistency is the property which states that a database must remain in a **consistent state** after any transaction. In other words, no transaction should have any **adverse effect** on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well **[1]**.
- **Isolation** – Isolation is the property which states that if more than one transaction are being **executed simultaneously** and in **parallel** in the database, all the transactions must be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction **[1]**.
- **Durability** – Durability is the property which ensures that a database should be **durable enough** to hold all its latest updates even if the system **fails** or **restarts**. If a transaction updates a chunk of data in a database and commits, then the database **must hold** the modified data. If a transaction commits but the system fails before the data could be written onto the disk, then that data must be **updated** once the system gets back into action **[1]**.

NoSQL databases embrace situations where the ACID model hinders the operation of the database. It relies on a **softer model**, appropriately known as the **BASE model**. NoSQL databases replace **ACID properties** with **BASE properties** (**B**asically, **A**vailable, **S**oft state, **E**ventual consistency). BASE model accommodates the **flexibility** offered by NoSQL and similar approaches to the management and curation of un-structured data.

II. The CAP-Theorem

At the **Symposium on Principles of Distributed Computing** in the year **2000**, **Eric Brewer** held a keynote talk about his experience with the recent changes in the development of distributed databases (**Brewer, Towards Robust Distributed System, 2000**). In the years before his talk, the size of data grew considerably, making it necessary to find more scalable solutions than the so-far existing **ACID-databases**. As a result new principles were developed, summed up under the **BASE-paradigm**. Brewer analyzed the **consequences** of this **paradigm change** and its **implications**, resulting in the **CAP-Theorem**, which he presented in his talk that was more of his **personal intuition** than an **actual proven fact**. However, the theorem had such a huge impact that many researchers picked it up from there and **two years** later the theorem saw the light of the day. It got formally proven.

In **2002**, **Seth Gilbert** and **Nancy Lynch** of **MIT** published a **formal proof** of Brewer's conjecture, rendering it a **theorem** in **Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services**.

Over the years, the CAP theorem has been constantly developed and slight adjustments have been made, most prominently by Brewer himself, who amended it in a later paper that some of the conclusions, while not wrong, could be misleading (Brewer, CAP twelve years later: **How the "rules" have changed, 2012**). However, the CAP-theorem still is one of the most important findings for distributed databases. It is widely adopted today by large web companies like **Amazon** as well as in the **NoSQL community [5]**.

A. CAP Acronym:

- **Consistency**: Consistency, in an informal sense, means that each server returns the right response to each request. In other words, a response that is correct according to the desired service specification, as there can be multiple possible correct responses. The meaning of consistency depends on the service.
  - **Trivial services**: Services that **do not require coordination** among the servers are termed as trivial services. For example, if a service returns the value of the constant ∏ to 100 decimal places, then a correct response is exactly the **very solution**. **No coordination** among the servers is required. Trivial services do not fall within the scope of the CAP Theorem.

- **Weakly consistent services**: Keeping in mind the inherent trade-offs implied by the CAP Theorem, a lot of effort has been given in the attempt to develop weaker consistency requirements that still provide useful services and yet avoid sacrificing availability. For example: **A distributed web cache**. Web content like images and videos are **cached on servers** that are placed in data centers throughout the world. Whenever a user requests a given web page, the content is delivered from a nearby web cache. Such a system guarantees a very high level of availability. The proximity of the cache servers to the end users ensure both that the responses are rapid, and also that network connectivity issues rarely prevent a response.
- **Simple services**: Simple services have **straight forward correctness requirements**. The semantics of the service are specified by a sequential specification, and operations are **atomic**. A sequential specification defines a service in terms of its execution on a **single, centralized server**: the centralized server maintains **some state**, and each request is processed **in order**, **updating** the state and **generating a response**. A web service is atomic if, for every operation, there is a **single instance** in between the request and the response at which the operation appears to occur. This is equivalent to saying: from the client's point of view, it is as if all the operations were executed by a **single centralized server**.
- **Complicated services**: Many real services have more **complicated semantics**. Some services cannot be specified by sequential specifications. Others simply require more complicated coordination, transactional semantics, etc. The same CAP trade-offs typically apply, but for simplicity not much focus is imparted on these cases **[6] [7]**.

- **Availability**: The CAP-Theorem requirement **availability** means **service guarantee**. It simply means that each request eventually receive a **response**. A **fast response** is always a welcome than a **slow response**, but as per CAP purpose, requiring an **eventual response** is sufficient to create problems. In most real systems, of course, a response that is **sufficiently late** is just as bad as a **response that never occurs [6] [7]**.
- **Partition tolerance**: The third requirement of the CAP theorem is that the service be **partition tolerant**. Unlike the other two requirements, this property can be seen as a statement regarding the underlying system. **Communication among the servers is not reliable, and the servers may be partitioned into multiple groups that cannot communicate with each other**. For general purposes, communication is simply treated as **faulty** if messages happen to be **delayed** and, sometimes, **lost forever**. As part of information it must be cleared that a message that is **delayed** for **sufficiently long** may as well be considered **lost**, at least in the context of practical systems **[6] [7]**.

The CAP theorem states that: **A distributed system can satisfy any two of these guarantees at the same time but not all three**.

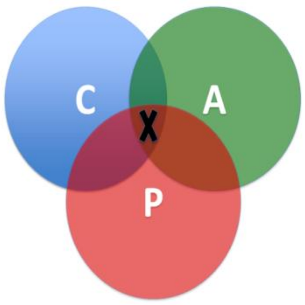


Fig 1: CAP Theorem

B. BASE Principles:

- **Basic availability**: The NoSQL database approach focuses largely on **availability of data**, even in case of multiple failures occurrence. It achieves this by using a **highly distributed** approach to database management. NoSQL databases do not maintain a **single large data store** and focus on **fault tolerance** of that instead they **spread data** across multiple storage systems with a **higher replication**. In case of a failure, disrupting access to a particular segment of data, which is highly unlikely, it does not necessarily result in a complete database outage **[2]**.

- **Soft state**: BASE model does away with consistency requirements of the ACID model almost completely. Its stores don't have to be **write-consistent**, nor do different replicas have to be mutually consistent over time. It is intended that the consistency after a transaction will not be in a **solid state** anymore but in a **soft state**. BASE model is of the view that data consistency is the **developer's problem** and should not be handled by the database **[2]**.
- **Eventual Consistency**: NoSQL databases have only one requirement regarding consistency. It requires that at some point of time in the future, **data will converge to a consistent state**. It however holds no guarantees regarding when this will occur. This principle is a complete departure from the immediate consistency requirement of ACID that prohibits a transaction from executing until all the prior transactions have completed and the database has converged to a consistent state **[2]**.

BASE focuses mainly on **permanent availability**.

III. Differences Between SQL and NoSQL Database

CRITERIA	SQL DATABASES	NoSQL DATABASES
Types	Only one type exists: **SQL database** with negligible variations of it.	A lot of types exist like, **Key-value** store databases, **Column-oriented** databases, **Document-oriented** databases, **Graph**-databases, **Object Oriented** Databases **[3]**.
Development	Developed at **IBM** in the earlier part of the **1970**s mainly to manipulate and retrieve data stored in IBM's original quasi-relational database management system, **System R**.	Developed in the late 2000s to cope with the short comings of SQL databases. It overcomes storing **large volumes of data**, storing **multi-structured data**, **replication of data**, **geo-distribution**, and overcoming **fault-tolerance** as well as storing **unstructured** and **semi-structured** data **[3]**.
Data Storage Model	Individual records (like customers) are stored as **rows** in tables and each **column** stores a **specific piece** of data about the records, an **attribute** (like customer ID, number of purchases) much like a **spreadsheet**. **Related data** are stored in **separate tables** and require **joining** when **complex queries** are needed to be executed.	Different types of NoSQL Databases have different types of data storing techniques. Details of which are given later in the article **[3]**.
Schemas	**Structure** and **data types** are **predefined**. In order to store a new type of record the **entire database** needs **alteration** and the database is taken **offline**.	NoSQL Databases are **mainly dynamic**, with some enforcing **data validation rules**. Records can be added while the database is **still online** and records of **dissimilar data types** can be stored together as and when needed. Only in **column-oriented databases** its bit of a challenge to add new fields dynamically **[3]**.
Scaling	Scaling is **vertical** or in other words a **single server** is made more powerful to cope up with increased data storage demand. SQL databases can be **spread** across **many servers** but a lot of **additional engineering** is required for that purpose and core relational features like **JOINs**, **referential integrity** and **transactions** are generally lost.	Scaling is **horizontal**. During data storage of extreme capacity the database administrator can add **commodity servers** or **cloud instances** with ease. The database automatically spreads data across the servers as necessary **[3]**.
Development Model	**Oracle database** is licensed under **Proprietary OTN Standard License**. **PostgreSQL** is licensed under **PostgreSQL License**. **MySQL** is an **open-source** RDBMS.	NoSQL databases are **open-source [3]**.
Data Manipulation	Manipulation can be done using specific language like **Select**, **Insert** and **Update** statements. For example: **SELECT fields FROM table WHERE…**	Manipulation can be done through **object-oriented APIs [3]**.

Consistency	Can be configured for **strong consistency**	Consistency depends on the **product**. Some NoSQL databases like **MongoDB** provide strong consistency. **Cassandra** offers eventual consistency **[3]**.
Examples	**MySQL**, **PostgreSQL**, **Microsoft SQL Server**, **Oracle Database**	**MongoDB**, **Cassandra**, **HBase**, **Neo4j [3].**

Table I: Comparison of SQL and NoSQL databases

IV. Characteristics of NoSQL Database

- NoSQL databases do not support ACID transaction properties as provided by RDBMS
- They do not have pre-defined schema.
- They can store huge volumes of data and have a flexible structure
- They support simple and flexible non-relational data models.
- No discontinuation of work takes place in case of any faults or failure in any machine.
- They can scale horizontally leading to high performance over many commodity servers.
- NoSQL provides **automatic sharding** and other features by default **[4]**.
- Maintenance of NoSQL databases is difficult.
- NoSQL databases lack a **standard query language** and many of them lack a **standard interface**. **[8]**

V. Principle of NoSQL Data Modeling

NoSQL provides **conceptual**, **general** as well as **hierarchical** data modeling techniques. Some of them include:

A. Conceptual Techniques:

- **De-Normalization**: De-normalization means **copying** of the same data into **multiple documents** or **tables** in order to **simplify** or **optimize query processing** or to **fit the user's data** into a **particular data model**. De-normalization **groups all data** that is needed to process a query in a **single place**. This often leads to accessing the **same data** in **different combinations** for different **query flows**. This requires **data duplication** which **increases** the **total data volume**. **Modeling-time normalization** and consequent **query-time joins** increases **complexity** of the **query processor**, especially in **distributed systems**. De-normalization allows storing of data in a **query-friendly** structure to simplify query processing. De-normalization finds wide applicability in **key-value stores**, **document-oriented databases** and **BigTable style databases [19]**.
- **Aggregates:** NoSQL provides **soft schema capabilities** as has been earlier said. **Key-value stores** and **Graph Databases** typically do not place **constraints** on values and they can be comprised of **arbitrary format**. It is also possible to **vary** a number of records for **one business entity** by using **composite keys**. For example: A user account can be modeled as a set of entries consisting of composite keys like **UserID_name, UserID_email, UserID_messages** and so on. If a user has no email or messages then a corresponding entry is not recorded. **BigTable models** support soft schema by virtue of a variable set of columns within a **column family** and a variable number of **versions** for one cell. **Document-oriented databases** are inherently **schema-less** though some of them do allow validation of incoming data using a user-defined schema. Soft schema allows formation of **classes of entities** with complex internal structures (nested entities) and to **vary the structure** of particular entities. This facilitates minimization of **one-to-many relationships** by means of **nested entities** and in turn **reducing joins**. It also does **masking** of **technical differences** between business entities and modeling of heterogeneous business entities using one

collection of documents or one table. Aggregation finds wide applicability in **key-value stores** and **document-oriented databases** and **BigTable style Databases [19]**.

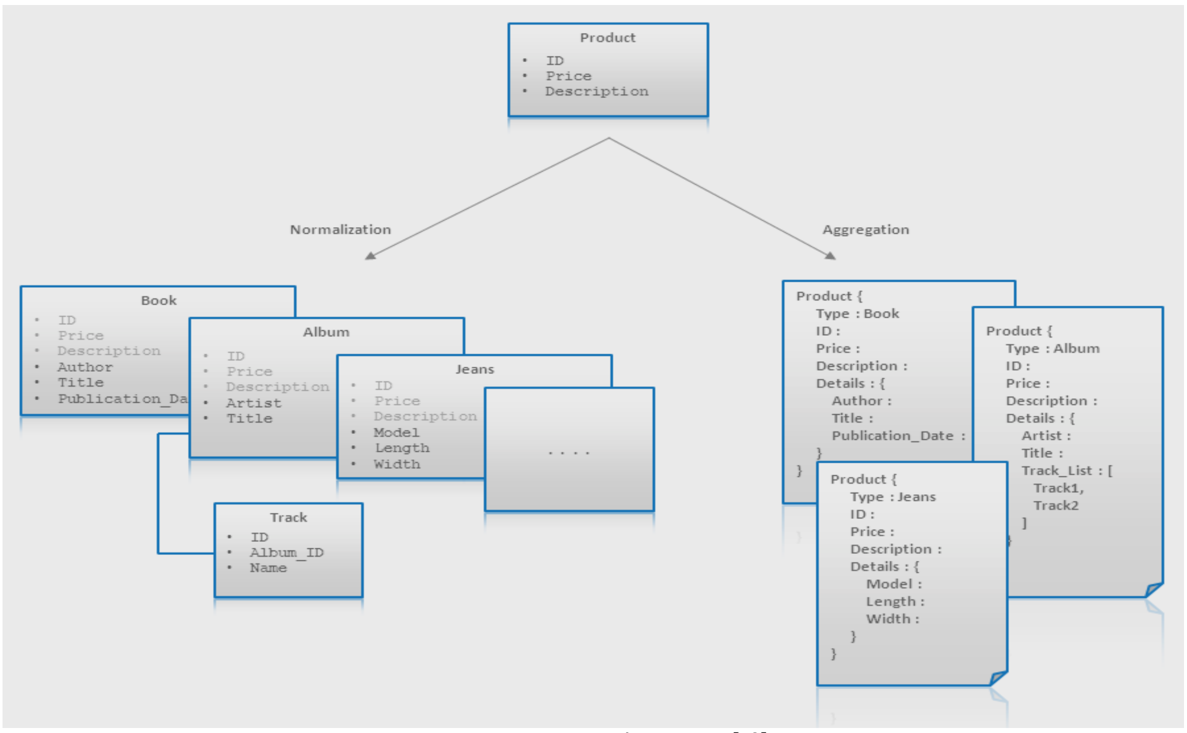

Product
ID
Price
Description
Normalization
Aggregation
Book
ID
Price
Description
Author
Title
Publication_Da
Album
ID
Price
Description
Artist
Title
Jeans
ID
Price
Description
Model
Length
Width
. . . .
Track
ID
Album_ID
Name
Product { Type : Book ID : Price : Description : Details : { Author : Title : Publication_Date : } }
Product { Type : Album ID : Price : Description : Details : { Artist : Title : Track_List : [Track1, Track2] } }
Product { Type : Jeans ID : Price : Description : Details : { Model : Length : Width : } }

Fig 2: Entity Aggregation [19]

- **Application-Side joins**: NoSQL solutions seldom support joins. As NoSQL databases are **question-oriented** so joins are handled at **design time** unlike in RDBMS, where joins are handled at **query execution time**. Query time joins always results in **performance penalty** but in many cases joins can be avoided by using **de-normalization** and **aggregates**, i.e. embedding nested entities. However in certain scenario joins have to be carried out like, **many to many relationships** are often modeled by **links** and **require joins**. **Aggregates** are often **inapplicable** when an **entity's internals** require **frequent modifications**. It is generally better to **keep a record** that something happened and **join the records** at **query time** as opposed to **changing a value**. It finds applicability in **key-value stores**, **document-oriented databases**, **BigTable-style databases**, **graph databases [19]**.

B. General Modeling Techniques

- **Atomic aggregates:** Majority of NoSQL solutions have **limited transaction support**. In certain situations **transactional behavior** can be achieved using **distributed locks** or **application-managed MVCC**, but it is common to model data using an **aggregates technique** to guarantee some of the **ACID properties**. **Relational databases** require **powerful transactional machinery** because **normalized data** typically require **multi-place updates**. **Aggregates**, on the other hand, allows storing a **single** entity as **one document**, **row or key-value pair** and update it **atomically**. Atomic aggregates as a data modeling technique is not a **complete transactional solution**, but if the store provides certain guaranties of **atomicity**, **locks** or **test-and-set** instructions then atomic aggregates is applicable. Atomic aggregates are applicable in **key-value stores**, **document databases**, **BigTable-style databases [19]**.

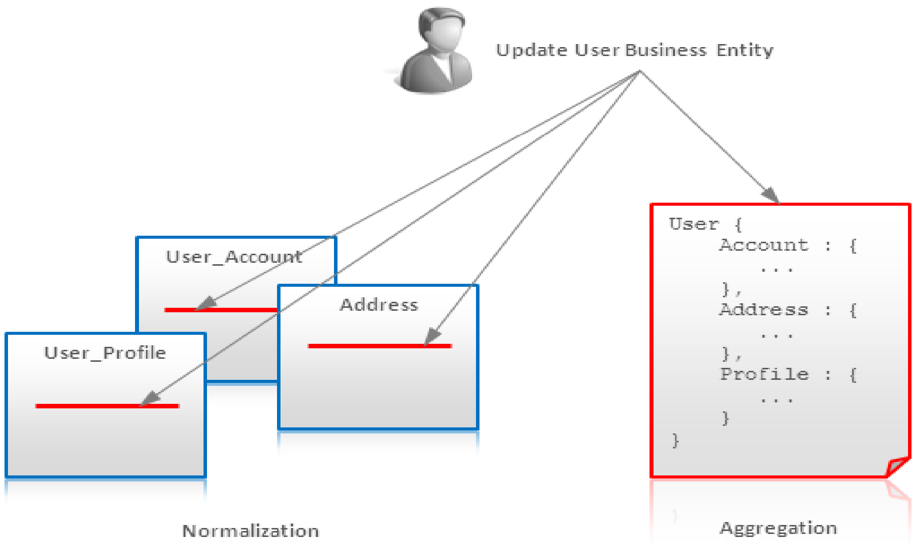


Fig 3: Atomic aggregates [19]

- **Dimensionality reduction**: Dimensionality Reduction is a technique that allows **mapping** of **multidimensional data** to a **key-value model** or to other **non-multidimensional models**. Traditional **geographic information systems** (GIS) use certain variation of a **Quad-tree** or **R-Tree** for **indexes**. These structures need to be updated **in-place** and are **expensive** to **manipulate** when data volumes are large. Alternatively the **2D structure** can be **traversed** and **flattened** into a plain **list of entries**. A **Geohash** implements this technique well. Geohash uses a **Z-like scan** to fill **2D space** and each move is encoded as **0** or **1** depending on direction. Bits for longitude and latitude moves are **interleaved** as well as moves. Geohash can estimate distance **between regions using bit-wise code proximity**. Geohash encoding allows one to store **geographical information** using **plain data models**, like **sorted key values** preserving **spatial relationships**. Applicability of dimensionality reduction can be found in **key-value stores**, **document databases and BigTable-style** databases **[19]**.

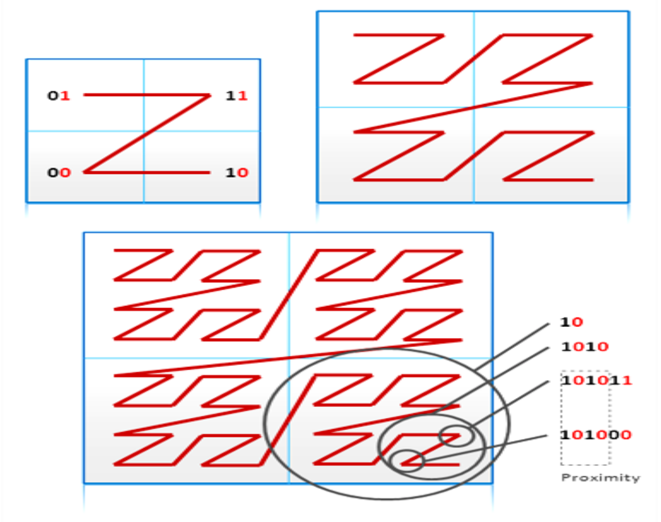
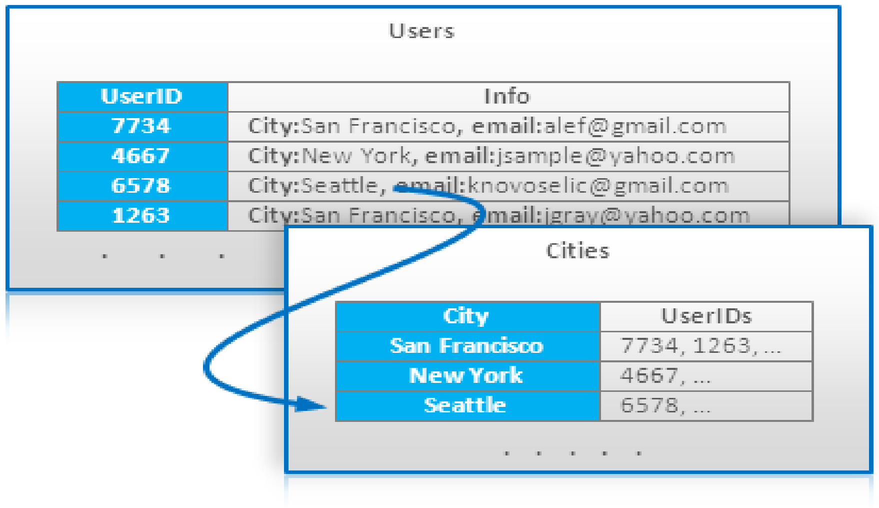


Fig 4: Geohash index [19]

- **Index Table**: Index Table is a simple technique that allows taking advantage of **indexes** in **stores** that do not support **indexes internally**. The most important class of such stores is the **BigTable-style database**. It creates and maintains a **special table** with **keys** that follow the **access pattern**. For example: There is a **master table** that stores user accounts that can be accessed by **user ID**. A **query** that **retrieves** all users by a **specified city** can be supported by means of an **additional table** where **city** is a key **[19]**.



UserID	Info
7734	City:San Francisco, email:alef@gmail.com
4667	City:New York, email:jsample@yahoo.com
6578	City:Seattle, email:knovoselic@gmail.com
1263	City:San Francisco, email:jgray@yahoo.com

City	UserIDs
San Francisco	7734, 1263, ...
New York	4667, ...
Seattle	6578, ...

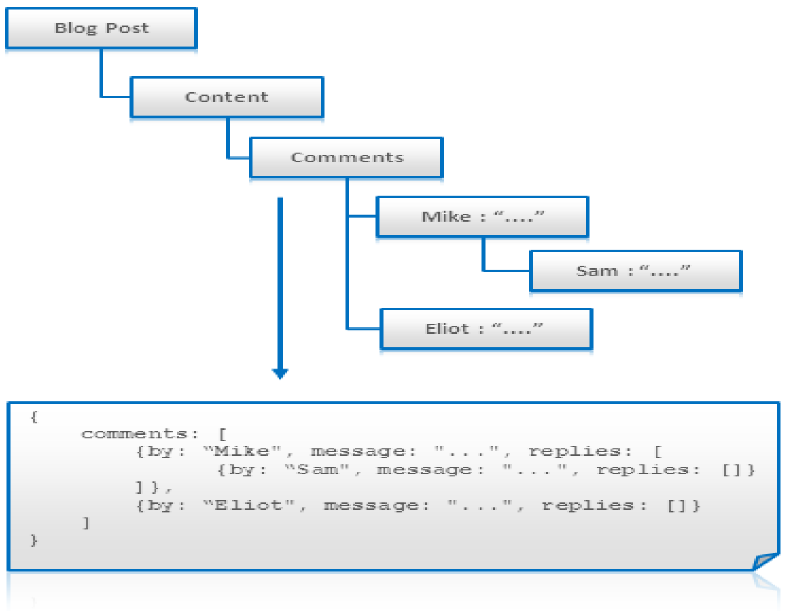
Fig 5: Index Table [19]

An Index table can be updated for **each update of the master table** or in **batch mode**. Either way, it results in an **additional performance penalty** and become a **consistency issue**. Index table finds applicability in **BigTable-style databases**.

Other general modeling techniques include **Enumerable keys** and **Composite key index [19]**.

C. Hierarchical Modeling Techniques

- **Tree aggregation**: Tree aggregation is a technique where **arbitrary graphs** with the help of **de-normalization** and **trees** can be modeled as a **single record** or **document**. Tree- aggregation is **efficient** when the tree is accessed **at once**, like an **entire tree** of **blog comments** is fetched to show a **page** with a **post**. However **searching** and **arbitrary accessing** to the **entries** happens to be **problematic** and **updating** is **inefficient** in most **NoSQL implementations** compared to independent nodes. Tree aggregation is efficient in **key-value stores** and **document-oriented databases [19]**.



```
{
    comments: [
        {by: "Mike", message: "...", replies: [
              {by: "Sam", message: "...", replies: []}
        ]},
        {by: "Eliot", message: "...", replies: []}
    ]
}
```

Fig 6: Tree aggregation [19]

- **Adjacency lists**: Adjacency lists are a simple way of **graph modeling**. Here each node is modeled as an **independent record** that contains **arrays** of direct **ancestors** or **descendants**. It allows searching for nodes by identifiers of their

parents or **children** and to traverse a graph by doing **one hop** per query. This approach is usually inefficient for getting an entire sub-tree for a given node, for deep or wide traversals **[19]**.

- **Nested sets**: Nested sets are a very good method for modeling **tree-like structures**. It is very much used in **relational databases** but is applicable to **key-value stores** and **document-oriented databases**. It stores **leaves** of the tree in an **array** and **maps** each **non-leaf node** to a **range of leaves** using **start** and **end indexes**. This structure is **pretty efficient** for **rigid data** as it has a **small memory footprint** and allows **fetching** of **all leaves** for a **given node** without **traversals**. **Inserting** and **updating** are however **costly** since **addition** of a **leaf** causes a **large updating** of indexes **[19]**.

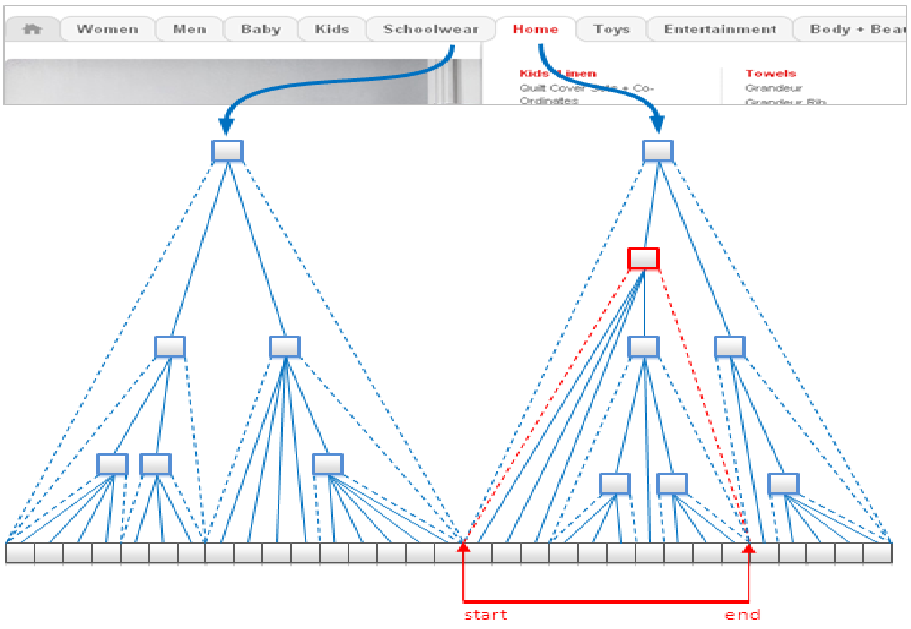


Fig 7: Modeling of eCommerce catalog using Nested Sets [19]

Other hierarchical modeling techniques include **Materialized paths**, **Nested documents flattening: Numbered field names**, **Nested documents flattening: Proximity queries**, **Batch graph processing [19]**.

VI. Types of NoSQL Database

NoSQL data modeling often takes off from the application-specific queries as opposed to relational modeling. Relational modeling is typically driven by the **structure of available data**. The main design theme is "**What answers do I have?"** NoSQL data modeling is typically driven by **application-specific access patterns**, or in other words the **types of queries** to be supported. The main design theme is **"What questions do I have?"** NoSQL data modeling more often than not requires a **profound understanding of data structures and algorithms** than relational database modeling. NoSQL Databases can be categorized into **five** types:

A. Key-Value Store Databases

The key-value data stores use **simple implementation methods**, yet are quiet an **efficient** and **powerful** model. It has a simple **application programming interface** (API). A key-value data store stores data in a **schema less manner**. The data is usually some kind of **data type of a programming language** or an **object**. The data consists of **two parts**: **the actual data** which is to be referred to as **value** and **a string which represents the key** creating in turn a **key-value pair**. Thus the data stores are similar to **hash-tables** where the keys are used as **indexes**, making it **faster** than RDBMS. Hence the data model is quite simple: **a map or a dictionary that allows the user to request the values according to the key specified**. 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 reduced. **High concurrency**, **fast lookups** and **options for mass storage** are important criteria looked into by the key-value stores. However, key-value data stores lack schema which makes it much more difficult to create **custom views** of the data.

Some practical uses of key-value data stores are situations where a user's session or a user's shopping cart needs to be stored to get details about their favorite products. Thus key-value data stores finds utilities in **forums**, **websites for online shopping** and the likes **[8]**.

Advantage: Key-value store databases have a **simple data model without relations or structures**. They are **quick** and **efficient** as far as data management in distributed systems is concerned **[10]**.

Examples: **Project Voldemort**, **RIAK**

Project Voldemort: Project Voldemort is an **open source database**, written in **Java**, created and used by **LinkedIn**. A **Voldemort cluster** can contain **many nodes**, each with a **unique identifier**. A **physical host** can run **multiple nodes** if needed, though LinkedIn maintains a **one-to-one mapping**. All nodes in the cluster have the **same number of stores** which is basically **tables** in the database. A **site-facing feature** can map to **one or more stores.** A feature with **group recommendations** will map to **two stores** for instance. **One store records** a **member ID** to the **recommended group IDs** and another **recording** a **group ID** to its **corresponding description**. Every store must have a list of **configurable parameters** including, **Replication factor (N)**: Number of nodes which each key-value tuple is replicated, **Required reads (R)**: Number of nodes Voldemort reads from, in parallel, during a **get request** before declaring a success, **Required writes (W)**: Number of node responses Voldemort blocks for, before declaring success during a **put request, Key/Value serialization and compression**: Voldemort can contain different **serialization schemas** for key and value, **Storage engine type**: Voldemort supports various **read-write storage engine** formats: **Berkeley DB Java Edition** and **MySQL**. Every node in the cluster stores the same **two pieces of metadata**: **the complete cluster topology** and **the store definitions [11]**.

Advantages:

- Efficient queries are possible to be executed and performance is very predictable.
- Distribution across a cluster is easy.
- Clean separation of storage and logic
- Object-relation miss-match does not exist **[12]**.

Disadvantages:

- No complex query filters.
- All joins must be done in code
- No foreign key constraints
- No triggers **[12]**.

B. Column-Oriented Databases

Column-oriented data stores use **hybrid row/column** like storage structure quite different from pure RDBMS. It possesses the concept of **column-by-column storage** of **columnar databases** and **columnar extensions** to **row-based databases**. **Column stores** are far from storing data in tables and store the data in **hugely distributed architectures**. In these stores, each key is associated with **one or more** attributes or columns. They store their data in such a manner that it can be **aggregated faster with reduced I/O activity**. They offer **high scalability** in data storage. The data which is stored in the database is based on the **sort order** of the column family. Column oriented databases are suitable for **data mining** and **analytic applications**, where the storage methods fits well for the common operations performed on the data **[8]**.

Advantages: **Aggregation** can be done very quickly in column-oriented databases, because all values of the same attribute are in **succession**. There are **no relations** between datasets **[10]**.

Examples: **Google BigTable**, **Apache Cassandra**

Google BigTable: Google BigTable is a **distributed storage system** for managing **structured data** in **Google**, built using **Java, Python, Go, Ruby**. It can reliably **scale to petabytes** of data and **thousands** of **machines**. Today it has achieved a very **wide applicability**, **scalability**, **high performance**, and a **high availability**. It is used by more than **sixty Google products** and **projects**, including **Google Analytics**, **Google Finance**, **Orkut**, **Personalized Search**, **Writely**, **Gmail**, **YouTube** and **Google Earth**. These products use BigTable for a variety of challenging workloads ranging from **throughput-oriented batch-processing jobs** to **latency-sensitive serving of data** to end users. BigTable does not support a **full relational data model** but provides clients with a simple data model that supports **dynamic control** over data layout and format, and **allows clients to reason** about the **locality properties** of the data represented in the underlying storage.

It is a **sparse**, **distributed**, **persistent multidimensional sorted map**. The map is indexed using a **row key**, **column key**, and a **timestamp (three dimensional mapping)**. Each value in the map is an **arbitrary array of bytes**.

The **row keys** in a table are **arbitrary strings** of **size up to 64KB**. Every read or write of data under a single row key is **atomic**, regardless of the number of different columns being read or written in the row, and is a **design decision** that makes it easier for clients to reason about the system's behavior in the presence of **concurrent updates** to the same row. BigTable maintains data in **lexicographic order by row key**. The row range for a table is **dynamically partitioned**. Each row range is called a **tablet**, unit of distribution and load balancing.

Column keys are grouped into sets called **column families** forming the **basic unit of access control**. 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. After a family has been created, any column key within the family can be used. The syntax of a column key name is: **family: qualifier**.

Each cell in a BigTable can contain multiple versions of the same data with these being indexed by **timestamp**. They are **64-bit integers**. Timestamps can be assigned by **BigTable itself**, represented as **real-time** in **microseconds** or **explicitly** assigned by **client applications**. Applications that need assurance about **no-collision** occurrence should generate their **unique timestamps**. Different versions of a cell are stored in **decreasing timestamp order** to facilitate the **most recent** versions being **read first**.

The **BigTable API** provides functions for **creating** and **deleting tables** and **column families**. It also provides functions for **changing cluster**, **table**, and **column family metadata** like **access control rights**. BigTable uses the distributed **Google File System** (GFS) to store **log** and **data files**.

The BigTable implementation has **three major components**. They are, **a library** that is linked into every client, **one master server** and many **tablet servers**. Tablet servers can be **dynamically added or removed** from a cluster to **accommodate changes** in **workloads**. The **master** is responsible for **assigning tablets to tablet servers**, **detecting the addition and expiration of tablet servers**, **balancing tablet-server load, controlling schema changes like table and column family creations** and **garbage collection of files** in GFS. Each tablet server manages a set of tablets. The tablet server **handles read and write requests** to the tablets that it has loaded, and also **splits tablets that have grown too large**. A BigTable cluster stores a number of **tables**. **Each table** consists of a **set of tablets**, and **each tablet** contains **all data** associated with a **row range**. Initially, each table consists of just **one tablet**. As a table grows, it is automatically split into **multiple tablets**, each approximately **100-200 MB** in size by default **[17]**.

Advantages:

- BigTable has **less response time** for queries as compared to RDBMS **[23]**.
- BigTable does not require **joins** and **normalization [22]**.
- **Data compression** can be done easily in BigTable **[23]**.
- BigTable has immense **scaling** capacity and offers high **availability [23]**.
- There is no **row length** limit **[22]**.
- **Unlimited** number of **connections** can be kept for each record **[22]**.

Disadvantages:

- BigTable is **not** an **open source** project **[23]**.
- Writing **query programs** are not possible as BigTable **doesn't support structured query language [23]**.
- **Lack** of strict **consistency** often leads to consistency problems **[23]**.
- **Data loss** can occur **[22]**.
- **Secondary index** is not supported **[22]**.
- **Lacks** advanced features for **data security [22]**.

C. Document-Oriented Databases

Document-Oriented databases store their data in the form of **documents**. Document-oriented stores facilitate **great performance** and **horizontal scalability** options. Documents inside a document-oriented database are quite similar to records in relational databases but are more **flexible** as they are **schema less**. The documents are of standard formats such as **XML**, **PDF**, **JSON** and others. In relational databases, a record inside the same database will have **same data fields** and the **unused data fields will be kept vacant**. In document-oriented databases only the attributes that are **really used** are defined **[8]**. An illustration of this would be:

Let us take one document for a person whose attributes are: **name**="John Doe" and **webpage**= "www.example.net".

Let us take another document with the following attributes: **name**="Jane Doe", **e-mail**=jane@example.net and **children** = ("Sarah", "Catherine", "Marcus").

In a SQL database, each one of these attributes must be defined in a **dataset** (**name**, **webpage**, **e-mail**, **child 1**, **child 2**, **child 3**). For John, **e-mail** and the **child1 child2 child 3** fields=NULL and for Jane, **webpage**=NULL. So there is an unnecessary overhead.

In document store databases it is defined as: **defined** (John: **name**, webpage; Jane: **name**, e-mail and children) **[10]**.

In 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. These keys may be a **simple string** or a **string that refers to URI** (Uniform Resource Identifier) or **path**.

Document stores are a bit more **complex** than key-value stores as they allow **encasing the key-value pairs** in document also known as **key-document pairs**. Document-oriented databases are used in applications where the data are **not stored in a table with uniform sized fields** but are stored as a **document having special characteristics**. Document stores are beneficial when the domain model can be split and partitioned across some documents. Document stores are not used if the database has a **lot of relations and normalization**. They can be used for **content management system**, **blog software** etc **[8]**.

Advantages: The **database schema is not fixed** hence the documents can have **arbitrary structures** and **attributes** associated with them **[10]**

Examples: **MongoDB, CouchDB**

MongoDB: MongoDB is a **schema-free**, **open-source**, **cross-platform**, **document-oriented database** written in **C**, **C++** and **JavaScript** and mainly run by **10gen Inc.** that also offers **professional services** around MongoDB. MongoDB databases reside in a **MongoDB server** that can host more than one such database which is **independent** and stored separately by the MongoDB server. A database contains one or more **collections** comprising of **documents**. Since MongoDB is **schema-free,** the documents within a collection can afford to be **heterogeneous** ones. On inserting the first document into a database, a **collection** is created **automatically** and the inserted document is **added** to this collection. MongoDB facilitates collection using **hierarchical namespaces** using a **dot-notation**.

The abstraction and **unit** of **data**, **storable** in MongoDB is a **document**, **a data structure comparable to an XML document**, a **Python dictionary**, a **Ruby hash** or a **JSON document**. MongoDB keeps documents by a **format** called **BSON** which is very similar to **JSON** but in a **binary representation** for **efficiency** and for supporting **additional data types** compared to JSON. BSON **maps readily** to and from JSON and also to various **data structures** in many programming languages.

Amongst the **data types** provided by MongoDB for document files are **scalar types**: boolean, integer, double; **character sequence types**: string, regular expressions, code; **objects** for BSON-objects; **object id**, a data type for 12 byte long binary values used by MongoDB and all officially supported programming language drivers for a field named **id** that uniquely identifies documents within collections; **null**; **array**; **date [13]**.

MongoDB provides high availability with **replica sets**, which contains two or more copies of the data, with each capable of acting as **primary** or **secondary replica** at **any time**. **All writes** and **reads** are done on the **primary replica** by **default**. Secondary replicas maintain a **copy** of the data of the primary one by virtue of **built-in replication**. In case of a primary replica **failure**, the replica set automatically conducts an **election process** to determine which secondary should become the primary. Secondary replicas can possibly serve **read operations**, but that data is only eventually consistent by default **[14]**.

MongoDB scales **horizontally** using **sharding**. The user can choose a **shard key** that determines how the data in a collection will be distributed. The data is **split** into **ranges** based on the shard key and distributed across **multiple shards**. MongoDB can run over **multiple servers** by **balancing the load** or **duplicating data** or in case **both** to keep the system running in case of **hardware failure**. MongoDB can be **easily applied** and **new machines** can be **added** to a running database **[14]**.

MapReduce can be used for **batch processing** of data and **aggregation operations**. For computing aggregation of query results, MongoDB provides the **count**, **distinct** and **group** operations that may be invoked via **programming language libraries** and can be **executed** on the **database servers**. Aggregation operators can be **attached together** to form a **pipeline,** like in **UNIX pipes**. The aggregation framework includes the **$lookup** operator which can join documents from **multiple documents** as well as **statistical operators** such as **standard deviation [13] [14]**.

JavaScript can be used in **queries**, **aggregation functions** like **MapReduce**, and sent **directly** to the **database** to be executed.

MongoDB supports **fixed-size** collections called **capped collections**. This helps maintaining **insertion order** and on reaching the specified size behaves like a **circular queue [14]**.

MongoDB is used by **MTV networks**, **Foursquare**, **The Guardian**. Projects like **CERN's Large Hadron Collider**, **UIDAI Aadhaar** (India's unique identification project) also use MongoDB. Sometimes it can be **unreliable** and **indexing** takes up lot of **RAM [8]**.

Advantages:

- MongoDB enables **horizontal scaling** by using **sharding**, thus distributing data across physical partitions to overcome hardware limitations. The data gets **automatically balanced** in the clusters.
- MongoDB provides **ACID properties** at the **document level**.
- It supports **replica sets**. If the primary server goes down, the secondary server becomes the primary automatically, without any human intervention.
- It supports the common **authentication mechanisms**, such as **LDAP** (Lightweight Directory Access Protocol), **AD** (Active Directory) and **certificates**. MongoDB enables connection over **SSL** (Secure Socket Layer) and the data can be **encrypted [20]**.

Disadvantages:

- There is no **join** operation.
- To achieve its performance and scalability MongoDB fails to meet up **transaction support**. MongoDB fits best if there happens to be a lot of data but the relation between them is **weak**. It is not fit for **strong** related data like bank account information.
- MongoDB using memory mapped file lets the **operating system** handle the caching. The size of the database is limited by **virtual memory** provided by the **OS** and **hardware**. On a **32-bit** machine, data as much as **4 GB** can be saved. **Serious data** must not be stored in MongoDB in a 32 bit machine. When the data **exceed** the capacity, insertions may **fail** without **warnings**. For production environment, **64 bits** is a must **[21]**.

D. Graph-Databases

Graph-databases **store data** in the form of a **graph**. The graph consists of **nodes** and **edges**, **nodes acting as the objects** and **edges acting as the relationship between the objects**. The graph also consists of properties related to nodes. A method called **index free adjacency**, where **every node** consists of a **direct pointer** which **points to the adjacent node**, is used. This method facilitates traversing **millions** of **records**. In graph databases, the main concern is the **connection between data**. They provide **schema less** and **systematic storage** of **semi structured data**. The queries here are **denoted as traversals** thus making them **faster** than relational databases. They are **easy to scale** and are **whiteboard friendly**. Graph-databases **support ACID properties** and are **rollback supportive**.

Graph-databases finds utility in **social networking applications**, **recommendation software**, **bioinformatics**, **content management**, **security and access control**, **network and cloud management** and others. If data is just in a **tabular format** with not much **relationship** between the data, graph databases are **ill-suited**. **OLAP support** for graph databases is not well developed. Graph databases do not facilitate **sharding** and are difficult to **cluster [8]**.

Advantages: Graph databases use **traversals** for querying data instead of **joins**, used in SQL databases. This is **cheaper** and **simpler**. Graph databases can be **horizontally** scaled with ease. They use **replications** of **vertices** and **edges**. **Graph partition** can be applied to the graph databases which is a technique that divides **a graph into pieces, such that the pieces are of about the same size and there are few connections between the pieces [10]**.

Example: **Neo4j**, **FlockDB**

Neo4j: **Neo4j** is a **graph database management system** developed by **Neo Technology, Inc**. It is an **ACID-compliant** transactional database with **native graph storage** and **processing**. **db-engines.com (http://db-engines.com/en/ranking/graph+dbms)** rates it as the **most popular graph database**, followed by **Titan**, **Giraph** and others. It is available in a **GPL3-licensed open-source community edition** having **online backup** and **high availability extensions** licensed under the terms of the **Affero General Public License**. Neo4j is written in **Java** and accessible by **software** written in other languages using the **Cypher Query Language** through a transactional **HTTP endpoint**. Neo4j comes in **3 editions**: **Community**, **Enterprise** and **Government**. It is **dual-licensed**, by **GPLv3** and **AGPLv3/commercial**. The **Community Edition** is **free** but is limited to running on **1 node** only due to the **lack of clustering** and is without **hot backups** (**hot backup**, also called a **dynamic backup**, is a **backup** performed on data even though it is **actively accessible** to users and may currently be in a **state** of being **updated**). The **Enterprise Edition**, that requires **buying** a **license** unless the **application** built on **top** of it is **open-sourced**, is free of the above **limitations** and allows **clustering**, **hot backups** and **monitoring**. The **Government Edition** is an **extension** of the **enterprise edition** and adds certain **government specific services** including **FISMA-related certification** (The FISMA or the **Federal Information Security Management Act of 2002 is an** act that recognizes the importance of information security to the economic and national security interests of the United States) and **accreditation support**.

Neo4j has binding for a number of languages like **Python**, **Jython**, **Ruby** and **Clojure**. It does not have binding for **.NET** as of yet. The access is recommended to be using a **REST interface**. It has **massive scalability** and can **handle graphs** of several **billion nodes/ relationships/ properties** on a **single machine**. It can **traverse** depths of **1000 levels** and beyond at **milliseconds** speed hence it is **many times faster** than a RDBMS. It has a **single** and **convenient object oriented API**. It does not support **sharding**. Neo4j must be used in **software** involving **complex relationships** like **social networking**; **recommendation engines** etc. It however misses certain graph calculations like **finding a common friend from a set of users** in a **social networking site**. Neo4j must be avoided if **relationships** do not exist among the data. Some of the **fortune 500 companies** that use Neo4j are **Adobe**, **Accenture**, **Cisco**, **Lufthansa**, **Telenor** and **Mozilla [15][8]**.

Advantages:

- Neo4j facilitates faster **retrieval**, **traversal** and **navigation** of more connected data.
- It represents **semi-structured data** very easily.
- Neo4j CQL query language commands are in **human readable format** and very easy to learn.
- It **does not require complex joins** to retrieve connected/related data as it is very easy to retrieve its adjacent node or relationship details without joins or indexes **[18]**.

Disadvantages:

- **Neo4j 2.1.3** latest version has a **limitation** of supporting number of nodes, relationships and properties.
- It does not support **sharding [18]**.

E. Object-Oriented Databases

In object-oriented databases the **data** or the **information** that is to be stored are represented as **objects** much like an object used in **OOP** (Object Oriented Programming). They can be taken as an **amalgamation of object oriented programming** and **database principles**. **Object-data stores** provide the **entire features** particular to **OOP** like **data encapsulation**, **polymorphism** and **inheritance**. The **class can be compared to a table**, the **objects, to a tuple** and the **class attributes to columns in a RDBMS**. Each object has an **object identifier** which can be used to **distinctly represent** that **object**. **Pointers** facilitate **direct retrieval** of **objects** thus making **access** to **data faster** in object oriented databases. Object-oriented databases **eases modern software development processes** and makes them **agile**. Object-oriented databases find application in scenario like **complex objects relationships**, **changing object structures** or if the **application defines members that are collections**. They find utility in **scientific research**, **telecommunication**, and **computer-assisted drafting** and others. The **drawbacks** of object oriented databases are that they are **highly attached to a specific programming language**. They are also **difficult to scale** once they **overreach their physical memory size**. Object-data stores must not be used in case of **data and relationships being simple [8]**.

Advantages:

- OODBs can define new **abstractions**; can control the **implementation** of these abstractions. The new abstractions can match the **data structures** needed for complicated tasks by virtue of new **abstract data types**. **OODB packages** allows creation of **new classes** with **attributes** and **methods**, have them **inherit attributes** and **methods** from **super classes**, **create instances** of the class each with a **distinct object identifier**, **fetch the instances** either **individually** or **collectively**, and **load** and **run** methods.
- OODBs facilitate **inverse relationships** to express a **mutual reference between two objects**. This process ensures **referential integrity** by establishing corresponding reference as soon as a reference is created. **Automatic deletion** too is possible by means of these references. An example of an OODB package that supports the automatic maintenance of **inverse relationships** is **ObjectStore**.
- The OODB model has an **OID** (Object Identifier) that is **generated automatically** by the system and ensures **uniqueness** of each object. Hence there is **no need for user-defined keys** in the OODB model. Besides the **OID cannot be modified by the application** and the concept of object identity provides a **separate and consistent notion of identity**, which is independent of how an object is accessed or modeled with descriptive data. Thus two objects are different if they have **different OIDs**, even if they have the same structures and the same values for all their properties. In the RDB model, each object identification is supported by **user-defined keys**. These objects are in turn considered the same object **[9]**.

Examples: **Db4o**, **ObjectDB**

Db4o: Db4o**,** broadly, **database for objects**, is an **embeddable**, **open source** object-oriented database designed for **Java** and **.NET** developers. It was **developed**, **commercially licensed** and **supported** by **Actian**. Db4o is **written** in **Java** and **C#**. It can run on **any operating system** that supports **Java** or **.NET**. It provides a **GUI** called **Object Manager Enterprise** (OME) which can be used for various purposes like **database connection**, **browsing databases**, **building queries** and **administrative functions**.

Db4o can be embedded in **clients** or other **software components invisible** to the **end user**. It needs no separate **installation mechanisms** and comes as a **single library file** with a **footprint** (the amount of memory or disk space required by a program) of about **670KB** in the **.NET version** and **1MB** in the **Java version**. It uses **TCP/IP** for **client-server communication** and allows **configuring port number**. **Communication** takes place via **messaging**. Client-server **performance** is however **dependent** on the **network bandwidth**. The **querying performance** can be **enhanced** using **lazy queries**. Db4o supports **automatic object schema evolution** for the **basic class model changes** like **field-name deletion** and **field-name addition**. **Complex class model modifications** like **field-name change** and **field-type change**, **hierarchy move** can be **automated** by **writing** small **utility update programs**. It provides **Native Queries** (NQ), which allows the **developers** to use common **object oriented programming languages** like **Java**, **C#**, **VB.net** to **access the database** instead of **string-based APIs** like **SQL**, and in turn **avoid** a **constant**, **productivity reducing context switch** between **programming language** and **data access API**. NQ also provide **type safety** (type safety is the **extent** to which a programming language **discourages** or **prevents type errors**) as well as remove the need to **sanitize** against **code injection**. **LINQ** (Language Integrated Query) support is **fully integrated** in db4o for **.NET version 3.5**. LINQ allows the **creation** of **object-oriented queries** of any complexity with the benefit of **compile-time checking**, **IDE intelliSense** integration and automated **refactoring**. LINQ can be used both against **relational** and **object data storage**, thus providing a **bridge** between them. It can also be used as an **abstraction layer**, allowing to **easily switching** the **underlying database technology**.

Some of the **Fortune 500 companies** that use db4o are **BMW**, **Bosch**, **IBM**, **Intel** and **Seagate [16][8]**.

Advantages:

- Db4o provides full **ACID** transactional capabilities [**24**].
- Deployable in **large volumes** without **local administration** [**24**].
- Db4o provides **easy installation** mechanisms.
- Db4o provides **fewer errors**, **better maintainability** and **software longevity [24]**.
- Db4o is a **company** and has a **community**, so they can **push** the **project** by **marketing**, **events** etc.

Disadvantages:

- Db4o lacks **full-text indexing** and perform poorly on **full-text search**.
- Lack of indexing for **string types**, **text based searches** can potentially be **very slow**.
- There is no **general query language** like **SQL** which can be used for **data analyzing** or by **other applications**. This prevents Db4o from being **flexible** in a **heterogeneous environment**.
- **Replication** cannot be done **administratively** and it needs **programming** an **application** to achieve it.
- **Deleted fields** are **never removed** but just **hidden** until the next **Defrag**.
- There is no **built-in support** to **import** or **export data** and to or from **text**, **XML** or **JSON files [16]**.
- It does not provide features like **referential integrity**, **OLAP tools** offered by SQL **[8]**.

VII. Conclusion

Looking at the way things are moving in the data production and providing analysis from them, there seems to be no better way than NoSQL. A small example of this would be the **IoT-generated data**. It presents **newer challenges** everyday and traditional relational databases fail to manage the **scale** of IoT data as well as many of them are **ill-equipped** to handle the **specialized nature** of IoT data sets, including **time series data**. Effectively **collecting**, **storing** and **analyzing** time series data is essential for harnessing IoT's power to help businesses gain **valuable insights**, **power digital transformations** and **drive more effective customer engagements**. Because of their capacity to handle huge volumes of data, NoSQL databases seem perfect to **weather** the coming data storm, which will come not only from IoT generated data but from various other sources. However NoSQL databases are **immature** and still many of the times they **crunch** during **extremely fast** and **extremely large datasets**, in no time. **Fault tolerance** in NoSQL is a great issue. It needs a serious look up for the future. Due to the **lack of SQL** like query language in them, users often find it difficult to quickly leverage and easily write **queries** for data analysis in them. The semantics are **unfamiliar** to the users. **Familiarizing** users with the **NoSQL methodologies** are essential for the upcoming days. A distributed NoSQL database with a perfect architecture will ensure high availability and high performance for both **read** and **write operations** during **peak loads**, even in case of a **node failure**. Many professionals suggest that if **transaction processing** applications generating a **few hundreds** of reports is required, **RDBMS** is a good solution. If **analytics** to the transactional applications are required then RDBMS is ill-suited. Consumers, business users do not need transactional nature of reports and to provide **intelligent reports** generated by mashed up data, NoSQL is the way to go.

ACKNOWLEDGEMENT

The authors are very much grateful to the Department of Computer Science, St. Xavier's College Autonomous (Kolkata) for giving the opportunity to do the research work in NoSQL database basic concepts, NoSQL data modeling techniques and the types of NoSQL databases.

References

1. http://www.tutorialspoint.com/dbms/dbms_transaction.htm, last visited: 13th August 2016
2. http://databases.about.com/od/otherdatabases/a/Abandoning-Acid-In-Favor-Of-Base.htm, last visited: 13th August 2016
3. https://www.mongodb.com/nosql-explained, last visited: 13th August 2016
4. International Conference on Recent Advances in Engineering Science and Management. PHD Chamber of Commerce and Industry, New Delhi. 30th August 2015. Title: A Comparative Study of NoSQL Data Storage Models for Big Data. Author: Ompal Singh. Assistant Professor, Computer Science and Engineering, Sharada University, India.
5. Brewer's CAP Theorem. Report to Brewer's original presentation of his CAP Theorem at the Symposium on Principles of Distributed Computing (POSC) 2000. CS341 Distributed Information Systems. University of Basel, HS2012. Author: Salome Simon.
6. Perspectives on the CAP Theorem. Authors: Seth Gilbert. National University of Singapore. Nancy A. Lynch. Massachusetts Institute of Technology.
7. The CAP Theorem and the design of large scale distributed systems: Part I. Author: Silvia Bonomi. University of Rome "La Sapienza", Great Ideas in Computer Science & Engineering. A.A. 2012/2013.
8. Types of NoSQL Databases and its Comparisons. IJAIS. Volume 5- No. 4, March2013. Authors: Ameya Nayak, Anil Poriya, Dikshay Poojary. Dept. of Computer Engineering. Thakur College of Engineering and Technology.
9. Journal of Object Technology. Published by ETH Zurich, Chair of Software Engineering. Volume-2. Number-4. July-August 2003. Author: Sikha Bagui. Department of Computer Science, University of West Florida.
10. NoSQL Databases. Author: Silvan Weber. Master of Science in Engineering. University of Applied Sciences HTW Chur, Switzerland.
11. Serving Large-scale Batch Computed Data with Project Voldemort. Authors: Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman and Sam Shah. LinkedIn
12. http://www.project-voldemort.com/voldemort/design.html, Last visited: 14th August 2016
13. NoSQL Databases. Lecture: Selected Topics on Software-Technology Ultra-Large Scale Sites. Authors: Prof. Walter Kriha (Lecturer), Course of Studies: Computer Science and Media (CSM) Hochschule der Medien, Stuttgart (Stuttgart Media University).
14. https://en.wikipedia.org/wiki/MongoDB, last visited: 14th August 2016
15. https://en.wikipedia.org/wiki/Neo4j, last visited: 14th August 2016
16. https://en.wikipedia.org/wiki/Db4o, last visited: 14th August 2016.
17. BigTable: A Distributed Storage System for Structured Data. Authors: Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
18. http://www.tutorialspoint.com/neo4j/neo4j_features_advantages.htm, last visited: 15th August 2016.
19. https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/, last visited: 12th August 2016
20. http://www.developer.com/db/mongodb-nosql-solution-advantages-and-disadvantages.html, last visited: 16th August 2016.
21. http://makble.com/the-advantages-and-disadvantages-of-mongodb, last visited: 16th August 2016.
22. http://mas.cmpe.boun.edu.tr/wiki/lib/exe/fetch.php?media=courses:cmpe473-fall-2012:bigtable.pdf Last visited: 16th August 2016.
23. BigTable, Dynamo & Cassandra – A Review , International Journal of Electronics and Computer Science Engineering. ISSN- 2277-1956, Authors: Kala Karun A, Subu Surendran. Sree Chitra Thirunal College of Engineering, Thiruvananthapuram
24. http://www.sunilgulabani.com/2013/01/database-for-objects-db4o.html, last visited: 20th August 2016.

Author(s) Profile

Kaustav Ghosh, passed M.Sc. in Computer Science from St. Xavier's College (Autonomous), Kolkata in 2016. He has already published papers on Cognitive Radio and Big Data in International Journals. Currently he is doing research work in Big Data Analytics, Data Science

Dr. Asoke Nath, is an Associate Professor in the Department of Computer Science, St. Xavier's College (Autonomous) Kolkata. Apart from his teaching assignment he is involved in various research fields such as Cryptography and Network Security, Visual Cryptography, Steganography, Image Processing, Mathematical Modeling of Social Networks, Li Fi technology, Big Data Analytics, Cognitive Radio, Data Science, e learning, MOOCs and so on. He has published more than 191 publications in Journals and conference proceedings. He is the life member of MIR Labs (USA) and CSI Kolkata Chapter.