CHAPTER 6

EFFICIENT NOSQL TECHNIQUES FOR CLOUD PARADIGM
6.1 Introduction

6.1.1 Technology Background of NoSQL Databases
6.1.2 Characteristics of NoSQL Databases
6.1.3 Classification of NoSQL Databases
6.1.4 Query Language Maturity

6.2 Cloud Systems that use NoSQL

6.2.1 Understanding Cloud Computing Landscape
6.2.2 Characteristics of Cloud Systems
6.2.3 Classification of Cloud Systems
6.2.4 Hoff’s Cloud Model
6.2.5 UCSB-IBM Cloud Ontology

6.3 NoSQL and Cloud Assimilation

6.3.1 The CAP Theorem
6.3.2 Data Management in Cloud Environment
6.3.3 Distributed Algorithms in Cloud Data Management
6.3.4 Sharing and Replication in Dynamic Environments

Summary

6.1 INTRODUCTION

6.1.1 Technology Background of NoSQL Databases

NoSQL as a database facilitates a structure around storage and retrieval of data. This data is organised in logical structures different from tabular relations often found in traditional relational databases. NoSQL database are also referred as “Not Only SQL”, is a methodology to data management and database design that's useful for fairly large chunks of distributed data. NoSQL can also be called a non SQL or non-relational database. Since the late 1960s, relational databases have existed but could not gain the title of NoSQL till the twenty-first century, emerged by the needs of companies
categorised as Web 2.0 such as *Amazon, Google and Facebook*. NoSQL databases were triggered due to the exponential growth of the internet and the rise of traffic generated on web 2.0 applications. Google published the *BigTable* research in 2006, and Amazon published the *Dynamo* technical paper in the year 2007. These databases were designed to meet a new generation of enterprise companies.

Conflicting to presumptions caused by its title, NoSQL never prohibited SQL. It’s a fact that some NoSQL systems are purely non-relational, while others simply avoid use of relational features such as fixed table schemas and operations on join. For example a NoSQL database might structure data into objects, key/value pairs instead of using traditional tabular relation.

The data arrangement used by NoSQL stores (e.g. wide column, key-value, graph or document) are not similar compared to default in relational databases, making some operations faster in NoSQL. The particular use case suitability of a NoSQL database is dependent on the business case it must solve. Occasionally the data arrangement used by NoSQL databases is also considered as more flexible when compared to relational database tables.

Based on CAP theorem, various NoSQL databases lack of consistency to provide availability, speed and partition tolerance. Blockers to the increased adoption of NoSQL databases include lack of standardized interfaces, use of low-level query languages and huge prior investments in legacy relational databases. Mostly, NoSQL databases lack true nature of ACID transactions, although a few stores, such as *Aerospike, MarkLogic, Google Spanner, FairCom c-treeACE* make up for such transaction based features. Based on the in depth analysis of NoSQL databases, we find they provide a concept of eventual consistency. Database changes are replicated to all nodes eventually (typically within a time frame of milliseconds) so queries on objects might not return latest and updated data points immediately or might result in reading stale data which is inaccurate, a problem called as stale reads. In addition to this, few NoSQL systems may include lost writes and other types of data loss. To the advantage of NoSQL stores provide features of write-ahead logging to avoid loss of data. For distributed transaction processing across various stores, an even bigger challenge is data consistency. It is difficult for both NoSQL and relational databases. In comparison to current relational
databases there are few systems that maintain both X/Open XA and ACID transactions standards for distributed transaction processing.

6.1.2 Characteristics of NoSQL Databases

Prior to getting into the characteristics of NoSQL Stores it is logical to list a few real world business case scenarios to implement a NoSQL database as their persistent component for backend data. One of the most basic and important driver is when an organization has a business scenario which is tough to solve using traditional relational database engineering. For organisations with a stable and mature domain business model supported by a traditional database providing all the required features, use of NoSQL is less probable leading to no change in its data storage mechanism. A few of the business cases are listed below that support the use of a NoSQL database rather than a traditional relational one.

- Cost of relational database is not scalable with increased traffic at an acceptable rate
- Data is provided in small updates varying over time hence the number of tables required to store a first normal form has become disproportionate to the data being held.
- Dynamic business model providing a huge set of temporary data that should not really belong in the main data schema. Various examples are that of retained searches, shopping carts, incomplete user questionnaires and site personalisation.
- Relational database is denormalised for performance issues or for convenience in data manipulation in web application.
- Dataset has large chunk of images or text and the column definition is a finite large object.
- Business needs to execute queries with your data set that involve complex hierarchical relations; Generic examples are business intelligence queries having an absence or missing piece of data. For the latter consider an example of “All females in Paris who have a pet dog and whose sister’s have not yet purchased a paperback this year”
In case of local data transactions, where durability is not a required characteristic, for example liking items on social websites, creation of transactions for such kind of events is overkill as the action fails the user will mostly repeat it until it works.

Even though there is no agreement on what exactly constitutes a NoSQL solution, the following set of characteristics is often attributed to them [90, 91]:

- **Flexible and Simple non-relational data models:** NoSQL databases provide easy to change schemas or are mostly schema-free and are engineered to manage a wide quality of data structures [92]. Existing data structures can be categorized into four broad categories: key-value stores, document stores, column-family stores, and graph databases.

- **Horizontal scalability over various commodity servers:** A few selective data stores have data scaling, while rest of them focus more with read and/or write scaling.

- **Provide increased availability:** Various NoSQL databases are engineered for highly distributed scenarios, and unavoidable partition tolerance. Hence, for providing high availability, these solutions choose to lack consistency in favour of giving more priority to availability. As a result of this NoSQL data stores are AP (Available/Partition-tolerant) data stores, whereas most RDBMs are CA (Consistent/Available).

- Generally, no support to ACID transactions is provided by NoSQL. They are referred as BASE systems (Basically Available, Soft state, Eventually consistent) [93]. Detailing the acronym, **Basically Available** refers to the data store being available all the time when it is accessed, even if its partially unavailable; **Soft-state** emphasizes that consistency is not the highest priority at the times and some inconsistency can be tolerated; and **Eventually consistent** highlights that after a certain time frame, the data store moves to a consistent state. However, some NoSQL databases are ACID compliant. For e.g. CouchDB.

NoSQL databases are designed to meet the following enterprises software development requirements which mark the reason behind the above characterise:

1. The need to develop with agility
2. Simplicity for easier development
To remain ahead in the current economy, enterprises must keep innovate – and with a fast pace than ever. Speed is a critical factor, but so is agility, since web 2.0 applications change far more quickly than legacy applications like ERP. Relational databases act as critical roadblock to the required agility, due to their rigid data model.

- **Flexibility for Rapid Development:** A core guiding principle of agile is pacing to evolving application requirements: If requirements change, the domain model tends to changes. Relational databases act as critical roadblock because the data model is rigid and defined by a static schema. Developers have to change the schema, or even worse, *schema change* request is forwarded to the database administrators. This dependency graph leads to slow pace or stops development. This is illustrated in figure 6.1.

![Figure 6.1: RDMBS – A Static Schema Blocks the Addition of New Attributes on Demand](image)

Comparatively a NoSQL document database promotes agile development, its schema-less nature and absence of static data modelling definition is the key. Data modelling definition is left to the developer writing the services. With NoSQL, the data model is created by the application model. Applications and services model data as JSON objects.

- **Simplicity for Easier Development:** Applications-Services model data as JSON objects (e.g., employee as an entity), multi-valued data points as collections (e.g., roles), and related data as nested JSON objects or collections (e.g. manager as an entity). The issue with relational databases is that read/write operations are done by *shredding*, or disassembling, and reassembling objects. We can term this as object-relational *impedance*
mismatch. Object-relational mapping frameworks are the workarounds for the same, problematic and inefficient at best. The data model of JSON is given in figure 6.2.

As an example, consider a resume mapping application. In the given scenario, the Application interacts with resumes and the user. Both resume and user are objects. An array for skills with a collection for positions is an internal part of the application. However, the user object needs to be shredded while writing a resume to third relational database. Application would require inserting six rows into three tables to store this resume, as illustrated in figure 6.3.

![Figure 6.2: JSON – The Data Model Evolves with Easy Addition of New Attributes](image)

Application would require reading six rows from three tables while reading a resume (as given in figure 6.4). A document-oriented NoSQL store performs reads/writes data JSON format. It is the de facto standard for producing and consuming data for mobile, web, and IoT applications. This standard eliminates the object-relational impedance mismatch and removes the overhead of ORM framework, hence simplifying the application development.

In contrast to traditional databases, objects are read and written with no shredding (An object itself can be read or written as a document with no alterations at service layer).

All of the above characteristics promote NoSQL databases as a suitable data management system for Cloud. Indeed, various databases are offered as a service today, such as
Amazon’s DynamoDB [94] and SimpleDB [95] are regarded to be NoSQL stores. Though, the absence of full ACID transactions can be a critical impediment to their growth in various mission-critical systems. For instance, Corbet et al., [96, 97] support the argument that it is more suitable to deal with performance problems due to the heavy use of transactions instead of trying to work around the absconded transaction support. However, the lack of interface standards, low-level query languages and the huge legacy investments made in relational SQL by enterprises are other major barriers to the adoption of fresh NoSQL data stores.

Figure 6.3: RDMBS – Applications that are shred Objects into Rows of Data Stored in Multiple Tables

![Diagram showing RDMBS applications]

Figure 6.4 (a): Applications that can Store Objects with Nested Data as Single Documents
6.1.3 Classification of NoSQL Databases

In this brief section of classification we categorize NoSQL databases based on the kind of storage model implemented by them, individually. There are multiple ways to categorize these databases which are over 220 in number. The logical reason behind the categorizing methodology is based on Data Model or Domain Driven Design being the focal point of data base technologies in the world of Web 2.0. Each type of NoSQL DB is described with an industrial business case example and its weak points. The description meets the basic issue of providing a data storage technology perspective, leverage guidance to researchers and practitioners to select the best-fit data store. The categories of NoSQL data stores are as follows:

- Key Value Stores
- Document databases
- Graph databases
- XML databases
- Distributed Peer Stores
- Object Stores

Detailed analysis of the available data stores:

**Key Value Stores:** They have an easy data model based on key-value pair logic that resembles an associative hash map or a dictionary [98]. As per the *Key-Value* pair
methodology, a key is unique identifier of the associated value and is used for storage and retrieval of the value in and out of data store. *Value* acts as an transparent object to the data store and used for storing arbitrary data point, thereby providing a schema-free domain-data model. The key-value stores can also be distributed on various clusters for faster processing, but are not suitable for implementing some business scenarios which basically use relations or structures. Applications that use relations, and structures are to be implemented in the client API along with key-value store. Due to some inherent nature of key-value object stores, they cannot handle querying or indexing. In these cases, the queries are executed only with the help of keys. The logical view Key-Value store is depicted in figure 6.5. The further categories of Key-value database stores are mentioned below:

1. **In-memory key-value stores** (maintains data in memory), like Redis [99] and Memcached [100]

2. **Persistent key-value stores** (stores the data on disk), such as Riak [101], Voldemort [102], and BerkeleyDB [103].

![Logical Structure of Key-Value Store](image)

**Figure 6.5: Logical Structure of Key-Value Store**

Examples: Redis, Voldemort Tokyo Cabinet/Tyrant, Oracle BDB

Typical applications: Caching of content

Strengths: Fast lookups

Weaknesses: Maintained data has no static schema
Practical application: For writing, a forum application with a home profile page that gives the statistics such as user's messages posted, etc. and the top ten messages by them. On hitting the page, it reads on the user's id from a key that is unique and fetches a string of JSON representing all the relevant information. A process running in the background recalculates for every 15 minutes and writes the information to the database independently.

**Document Databases:** The focus point concept of a document database is notion of a "document". Each implementation of document-oriented store differs on the details of its definition, in a common, they are all based around the logic that “documents encapsulate and encode information (or data) in standardized encodings or formats. Most common encodings used includes YAML, XML, JSON and binary forms of BSON. Similar to a unique primary key, documents are referred in the store via a key that points to that document. An important highlighting characteristic of a document-oriented store is that the store provides a query language or API that fetches documents based on their contents, in addition to the key fast lookups executed by a key-value database. The typical example for normalized document model is given in figure 6.6.

![Normalized Document Model](image)

**Figure 6.6:** Normalized Document Model
Various implementations offer multiple ways of grouping and/or organizing documents:

- Collections
- Non-visible metadata
- Directory hierarchies
- Tags

In comparison with traditional relational databases, for example, tables could be referred analogous to collections and records analogous to documents. Yet they are not the same. Each record in a traditional table has the exactly the same sequence of fields, whereas in a collection documents may have fields which completely different.

Examples: MongoDB and CouchDB

Typical applications: Web applications

Strengths: incomplete data tolerance

Weaknesses: No standard query syntax and query performance.

Practical application: For creating a web application, that maintains profiles of refugees. We need to record details of each child with circumstances that vary tremendously, for example, a lost child in the camp may know their last name and may not know their parent's last name. Eventually a local may claim to recognise the person and offer some extra information, which is required to store, but unless verified the record, it may be treated sceptically.

**Graph Databases:** As the name suggest, Graph stores use graphs as the data model and originated from graph theory. Graph is a mathematical methodology that represents an object set, known as graph nodes or vertices, and the edges (or links) interconnect these vertices. They maintain relationships among many graph data nodes which is a completely different data model than the column-family, key-value and document stores. In graph stores, the edge and nodes also have discrete properties entailing of key-value pairs. They are dedicated in processing highly organized data and hence are very proficient in traversing relationships across various entities. Different applications of graph databases include social networking applications, pattern recognition,
recommendation systems, solving path locating problems, dependency analysis. They are relevant in scenarios such as pattern recognition, recommendation systems, social networking applications, dependency analysis, and many more. An example for normalized weighted graph is illustrated in figure 6.7.

**Figure 6.7:** Normalized Weighted Graph

Examples: Infinite Graph, Neo4J, InfoGrid

Typical applications: Recommendations and Social networking

Strengths: Graph algorithms e.g. connectedness, n degree relationships, shortest path, etc.

Weaknesses: Has to traverse the complete graph to conclude a definitive answer.
Practical application: Web application that needs social networking is most apt to a graph database. These matching principles can be protracted to any application where we need to comprehend what people are buying, doing or enjoying so that we can suggest further things for them to buy, do or like. Any time it is required to reply the query along the lines of *which malls, do people who are over-50, like skiing and have visited Florida dislike?* a graph database helps to answer such queries.

**XML Database:** An XML database is a software system that is data persistence and allows data to be quantified, and sometimes stored, in XML format. These data can be fetched, converted, extracted and sent back to a calling system. XML stores are a savour of document-oriented databases. The following example gives the type query in IBM DB2 SQL.

```sql
select id, vlume, xmlquery("$j/name", parse jnural as "j") as name
from dbo.journals
where xmlexists("$j[licence="abhishek_crc_publishrs "]", passing jnural as "j")
```

Example of XML Type Query in IBM DB2 SQL

Examples: Mark Logic, Exist, Oracle,

Typical applications: Publishing

Strengths: Schema validation and Mature Search technologies

Weaknesses: re-write is easier than updating the documents, No real binary solution,

Practical application: Any publishing entity that uses bespoke XML formats to yield print, web and eBook varieties of their articles. Editors need to rapidly search either semantic or text sections of the mark-up. They persist the XML of completed articles in the XML store and bind it in a readable-URL REST web service for the document
production systems. Workflow metadata attributes (e.g. stage of a manuscript) is stored in an isolated RDBMS. When system-wide alterations are required, XQuery re-writes bulk updates all the documents to match the new template.

**Distributed Peer Stores:** They are basically column family stores that are designed to manage large amount of data distributed over several servers. Similar to Key-Value stores, keys are used as unique binding primary keys. Moreover, the key points to numerous columns of the database. They are also some times referred as derivative of Google’s *Bigtable* [104], where the data is placed column wise. In *Bigtable*, multiple rows of dataset are uniquely identified by a *primary key*. In this case also, the set of column keys of a particular row makes it similar to key-value store and every column key represents a name-value pair. *Bigtable* concepts are directly implemented by the Hadoop HBase [105], whereas DynamoDB [106] and Amazon SimpleDB [94] are slightly different than the Bigtable. Only one set of column name-value for each row with no column families is used in SimpleDB and DynamoDB. Additional functionality of super-columns formed by grouping many columns together is used in Cassandra [107]. An illustration of logical structure of a distributed peer data store is shown in figure 6.8.

*Figure 6.8: Logical Structure of Distributed Peer Data Store*
Examples: Riak, Cassandra, HBase

Typical applications: Distributed file systems

Strengths: Good distributed storage of data and Fast lookups

Weaknesses: Extremely low-level API

Practical application: A news application site where any section of content: comments, author, articles, profiles, can be voted on and a not obligatory comment provided on the vote. We insert one store per user and one store per section of content, using a UUID as the key (producing one for each section of content and user). The user's store maintains each vote they have ever casted while the content vessel holds a copy of each vote that has been casted on the section of content. A overnight batch job is used to detect content that users have voted on, we identify a list of content for each user that has excessive votes but which they have not voted on. We then push this list of suggested articles into the user's vessel.

Object Stores: An object store (also known as OODBMS) is a database management system that represents information as objects, as used in object-oriented programming. Object databases are poles apart from relational databases. They are a hybrid version of both approaches.

Object-oriented database management systems merge database competencies with object-oriented programming language abilities. OODBMSs allow object-oriented programmers to produce a product, persist them as objects, and modify or replicate prevailing objects to generate new objects within the OODBMS. Because the database is assimilated with the programming language, the developer can persist consistency within single environment, in such a case, both the OODBMS and the programming language maintain the same data model of representation. By way of contrast, Relational DBMS projects maintain a clearer division between the database model and the application. The logical structure of object store is given in figure 6.9.
Examples: ObjectStore, GemStone, Polar, Oracle Coherence, db4o

Typical applications: Finance systems

Strengths: Low-latency ACID, mature technology, Matches OO development paradigm

Weaknesses: Limited batch-update options and querying

Practical application: Global company has a monoculture of trading development and wants to have trades done on desks in New York and Japan goes through a risk inspection process in London. An object signifying the trade is pushed into the object store and the risk check listener is listening for modification or appearance of trade objects. As the object is duplicated into the local European space, the risk check listener reads the Trade and evaluates the risk. Post this operation it rewrites the object to alert of a trade approval and sends an actual request of trade fulfilment. The trader's client listener waits for changes to be published on objects that contain the trader's id and appends the local information of the trade packet in the client. Hence, indicating to the trader of an approved trade.
6.1.4 Query Language Maturity

The abilities of querying by different data stores play a very important role while selecting them for an application. Many commercial data stores provide many APIs that act as an interface to interact with applications which directly makes an impact on the choice of a data store. A detailed comparison of mostly used NoSQL solutions is listed in table 6.1.

**Table 6.1: Tabulation of APIs and Services Offered by Prominent NoSQL Solutions**

<table>
<thead>
<tr>
<th>NoSQL Data Stores</th>
<th>Querying Capabilities of each NoSQL Data Store</th>
<th>Other API</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>API</td>
<td>Map Reduce</td>
</tr>
<tr>
<td>Key value stores</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redis</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Berkely DB</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Riak</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Column family stores</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cassandra</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Hbase</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Amazon SimpleDB (Amazon service)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Document stores</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MongoDB</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CouchDB</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CouchBase Server</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Graph database</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neo4J</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Hyper GraphDB</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Allegro Graph</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
In some cases, key-value database may not process querying formulated based on the contents of the value of the pair, as these values are not transparent to the data store. In contrast, a document database is capable of content based querying as its data model facilitates indexing and querying the document contents. Even these data objects assist MapReduce framework, which is basically a parallel processing programming model mean for large datasets [108]. It is well suited for distributed data processing on a cluster of servers. Similarly, SQL-like querying is a preferred option due to its wide usage in last decade and the same is also adopted in NoSQL. Hence, few players in the market like MongoDB [109] provide many alternatives like CQL [110] Cassandra and SparQL [111] by Allegro Graph [112] and Neo4j.

6.2 CLOUD SYSTEMS THAT USE NoSQL

6.2.1 Understanding Cloud Computing Landscape

Computing on cloud is basically on-demand and ubiquitous. It has convenient network access to a distributed pool of resources (e.g., network, servers, storage, applications, and services) that can be provisioned and released quickly with minimal administrative effort [113]. It represents a typical computing platform is regarded as a cloud, through which individuals and businesses organizations can avail the services of cloud applications irrespective of their geographic locations on demand [114]. The typical corporate cloud scenario is depicted in figure 6.10. Some features of the cloud-computing to facilitate NoSQL are list below[113].

- providing a user to access cloud source services with no human interaction as and when required;
- creation of pool of services and resources to serve various consumers;
- rapid, elastic and automatic provisioning of resources;
- network access to clients for better access of cloud services.
- Measured in which resource usage is controlled and monitored.
On birds eye view cloud computing mandates to provide ease of access through the web, less expenditure for infrastructure to startup software ventures, higher scalability, optimal cost, and reduced maintenance and business risks expenses [114].

6.2.2 Characteristics of Cloud Systems

Some new requirements in the context of data management are essential in cloud environments and the same are listed below:

- Better scalability and performance in comparison to contemporary applications in throughput of data storage solutions.
- To avoid heavy traffic, elasticity of cloud resources and applications is essential to carry out real time applications.
- Heterogeneous servers of different hardware and software configurations need to be considered while running applications.
- To cope up with server or application failures, the cloud software should be fault tolerant.
• As data security and privacy is utmost important, cloud solutions must provide essential security features so that data theft does not occur.
• High availability of computing resources is required to ease the migration of cloud applications to avoid downtime.

3.2.3 Classification of Cloud Systems
In past, several cloud-computing experts developed three sets of the cloud and popularly known as the SPI model, which is denoted as below:

• [S]oftware of the cloud
• [P]latform of the cloud
• [I]nfrastructure systems of the cloud

Cloud Software Systems: The first subset identifies applications developed and deployed on the Internet for the cloud, referred to as software as a service, wherein the user of this subset of systems is the end user. Such applications are majorly works on browser having pre-defined features and accessed on fee per usage basis. Resource providers define the fee in the contract. Some examples of SaaS are Google Apps like Google Docs and salesforce CRM system, SaaS is referred by end users as a smart alternative to desktop applications for various reasons.

Cloud Platform Systems: It is also known as platform as a service where the cloud supplier facilitates the applications developers with necessary APIs and IDEs for developing cloud based applications. The developers who build cloud platform systems use few APIs to build and then deploy them and finally test and fine-tune their applications. Google’s App Engine is one of the examples of systems in this category. PaaS services provide an APIs for measuring and billing information, which allows software developers to readily develop bill on the consumption of resource with respect to their business model. It helps to enforce and integrate the end-user and developer relationships.

Cloud Infrastructure Systems: In this SPI based classification model, the infrastructure resources like storage, compute and communication services are provided in a non-rigid manner which are also known as infrastructure as a service. Here, all the resources are
assumed to be schedulable which are enabled by virtualization at OS level. Hence, OS virtualization permits IaaS providers to efficiently manage the utilization of hardware resources.

### 3.2.4 Hoff’s Cloud Model

Inspired Christofer Hoff the SPI and UCSB-IBM cloud ontology team organized an online discussion and collaboration between various cloud-computing experts to develop ontology upon these models. Figure 6.11 depicts the Hoff’s cloud model. This model emphasizes on the basic three layers of cloud computing i.e., IaaS, PaaS, and SaaS. The model further divides the *IaaS* layer to sub-layers such as data center provides, hardware and network components. The next layer is an abstraction that bridges the hardware through grid, VM monitors and cluster utilities. Core connectivity and delivery is the next component, offering the various services like, authentication services and DNS services enabling the systems utilizing the IaaS layer.

Hoff’s cloud model’s abstraction component dealing with issues like connecting the resources and delivery of services are interleaving; hence, these layers are interdependent on services offered by one another. APIs offer the management services via an interface layer on the cloud. *GoGrid CloudCenter* API is one of the example systems for implementing such API sub-layers. PaaS layer in Hoff’s model is constituted as one of the important layer offering the basic integration services. This sub-layer like database, authentication and querying services provides several other services. The top layer of Hoff’s model is the SaaS layer, which is auxiliary divided into various components.

The real actual data and its constituents are stored in structured meta data format in the cloud which can be easily accessed for various purposes. SaaS layer is divided as embedded, native and web applications. Native applications are desktop application that use cloud service, Web applications are cloud based which read from the browser. At last, embedded applications are basically cloud based applications that embed one in another. The presentation sub-layer is used for video, data and voice presentation that are essential to recognize various forms of cloud based data presentations.
As detailed in Figure 6.11, Hoff’s model covers all the essential components of the cloud. The amplified elaboration shows more challenges and issues related to cloud computing; but addressing all makes the architecture more complex. However, the major landmark models of cloud as discussed in this chapter are seems to be complementary and shows different viewpoints of newer technologies related to cloud.

### 6.2.5 UCSB-IBM based Cloud Ontology

Cloud ontology as developed by UCSB-IBM in an endeavor to understand the cloud computing topography is one of the most referred cloud reference architecture. The objective of this reference architecture is to make a proper evaluation the technologies that support cloud computing and to provide an academic viewpoint of the same. Here, the basic ontology is evolved from SOA and the same is categorized at various levels of cloud. SOA composition makes the ontology assembled and provides a coordinated
service to each other in a form of service composition. In this viewpoint, cloud services can also be composed of multiple other cloud services.

As per the principle of compensability, the cloud model of UCSB-IBM is classified into five layers. In this model, every layer incorporates different services. The cloud services are at the same level of each layer at their corresponding level of abstraction, for example, all cloud software environments target application developers, whereas the applications themselves are intended for end clients. Hence, they are categorized in a different layer other than cloud applications. In this cloud ontology model, the whole architecture is modeled as a cloud stack, where each lower layer offers service to the upper layer.

In some aspects, both SPI classification and UCSB-IBM cloud ontology are similar, except that the IaaS layer is divided into resources, storage and communications sub-components. Brief explanation of figure 6.12 of the five layers is as follows:

1. **Applications (SaaS):** The first layer is similar to the SPI model and is referred as application layer of cloud. It is the top layer that directly interacts with the cloud consumers. Usually, the cloud consumers use the services being offered by this
component via browser. The consumers who use the services of this layer will be charged as per usage.

2. **Cloud Software Environment (PaaS):** This is the operating system or the middleware of the cloud, from where the consumers use its services to deploy their applications and also the IaaS resources are managed via this layer.

3. **Cloud Software Infrastructure:** This is the bottom most layer that provide hardware infrastructure to the cloud users and it is almost similar to SPI ontology. This layer is categorized into three sub-components as mentioned below.
   a. Computational resources: The computing power or the CPU power is the most frequently used hardware infrastructure from which the cloud users can get their computing power. This layer also hosts VMs to enable OS virtualization so that the computational resources are available on demand for the cloud users in a highly flexible way. It also shields physical infrastructure at the data centre from the cloud users.
   b. Storage: As storing data is a primary use of many applications, data storage is served as an infrastructure for the consumers to store their data at geographically remote data centres. The data can be fetched on demand from authorised users. This is provided as data-storage as a service to all the cloud users.
   c. Communication: Cloud vendors also provide network communication as service along with guaranteed QoS. Communication bandwidth is provisioned dynamically along with virtual overlays for traffic isolation, ensured network security, dedicated communication bandwidth, network monitoring, encryption and desirable message delay limits.

4. **Software Kernel Layer:** Like the traditional OS kernel, cloud systems also maintain a separate layer for software management of physical resources and the same is known as software kernel layer. This layer is also responsible managing and converting all the VMs that are in use. The typical components of this layer include VM monitors, hypervisors, OS kernel, clustering middleware etc. Typically, cluster/grid computing applications are installed at this layer that make it possible to run on the multiple connected clusters.
5. **Cloud Hardware/Firmware**: It is the bottom most layer of UCSB-IBM ontology meant for dealing with hardware switches, servers, and other physical hardware of the cloud. Big corporate cloud consumers and enterprises with huge IT requirements usually go for sub-renting hardware infrastructure as services. For example, in 2004 both IBM and Morgan Stanley’s entered into an agreement to use their hardware as a service.

### 6.3 NoSQL AND CLOUD ASSIMILATION

#### 6.3.1 The CAP Theorem

To facilitate storage and processing of enormous datasets, a widely used technology is to partition the data and store them at multiple servers. These partitions may be replicated at several places to increase data availability and also to avoid server failures. Modern technologies like BigTable, Cassandra use different partitioning methods that optimize for better availability and scalability. Nonetheless, these technologies which are basically stored in multiple connected data stores face some bottleneck as discussed in CAP theorem. Among the three properties of consistency, availability and partition tolerance, two can always be contented by any data systems on network at once.

![Figure 6.13: The CAP Theorem Representation](image)

[93]
As interpreted in CAP (Figure 6.13), consistency deals with having only one updated copy of the data unit at any point of time, and hence it slightly differs from ACID properties of database systems. In case of ACID, the consistency means the capability of having consistent state of database at any point of time throughout the transactions. Incase of availability, the system is required to make the data available for service at any point of time. At last, the partition tolerance means it is the system’s ability to tolerate the data partition at the network, so that the partition should not degrade the system throughput. Straight forward meaning of of CAP theorem is to make sure that when the data is distributed among multiple nodes using some partitioning technology, one has to ensure that the data is consistent. In case of rejection of write request by the data store in both partitions it implies that the data is consistent, but not available. In contrast, if both (or one) of the partitions admits write requests, then it is is available, but inconsistent hypothetically.

Even though the conclusion of CAP theorem is relatively simple in its conclusion it has its own critical implicaitons because of heterogeneously distributed data stores which are mainly trade-offs centric. To be more precise, the issues and challenges of traditional RDBMS in managing big data and the restrictions of CAP theorem on the use of distributed systems give rise to the use of techniques like No or New SQL.

6.3.2 Data Management in Cloud Environments

Data Consistency: Scalability as being the main moto of NoSQL movement compels distributed systems to resist over system failures, coordination, managing distributed resources and many other issues that typically occur in distributed environments. Even though NoSQL is basically different from traditional distributed databases, it attracts new challenges over the existing ones in distributed systems. Hence, newer protocols and algorithms have to be investigated for a robust NoSQL technology. Recent advances in NoSQL technologies emphasize on improving practical efficiency by building system of relevant databases. In the past, NoSQL focused on tradeoffs among performance, fault-tolerance and consistency to serve highly available applications with low latency over geographically distributed data. But, majority of these tradeoffs focus mainly on consistency of data and hence we discuss more on replication and repair of data in the
following section. But it is quite obvious that in heterogeneous geographically distributed environments with network partitions the delays are quite common and hence it is not possible to maintain high availability without compromising on consistency. Since, different sections of database operate autonomously in isolation due to network partition, it is not possible to provide consistency among database. The data consistency model in distributed systems is depicted in figure 6.14.

![Data Consistency in Distributed Computing](image)

**Figure 6.14:** Data Consistency in Distributed Computing

As consistency is more complicated to achieve, many scenarios of consistency is discussed below in detail.

- **Read-Write consistency:** Here, the main objective of database is to minimize a replica conjunction time (in simple words how long does it take to propagate to all
replicas with a update) and promise eventual consistency. Besides these weak promises, one can be involved in stronger consistency properties:

- **Read-after-write consistency**: Here, once data item X is operated, it is always observed by its successive operation read on X.

- **Read-after-read consistency**: In case if an application read the value of a data item X, then any successive operation which reads X always result in same or a more recent value.

- **Write-Write consistency**: In case of write-write conflicts that occur in a case of database partition, the database should either handle this conflict by its own or it should promise that parallel writes are not handled by different partitions. From this viewpoint, a database can provide several consistency models:

  - **Atomic Writes**: An API where the write request is an individual and independent atomic assignment of a value. But, it may attract write-write conflict. To avoid the same, it select the *most recent* version of all the entities. This makes all nodes end up same version of data. Also, the network failures and delays make unrelated order of updates which are further altered by the system, hence data version can be mentioned by a application-specific metric timestamps. This method is used in Cassandra.

  - **Atomic Read-modify-write**: Database usually receives do read-modify-write sequence request in lieu of unrelated atomic writes. If two users read the same version of data, transform it and write back in-parallel, the most recent update will silently override the first one as per the model of atomic writes.

A NoSQL database can provide the following approaches to avoid the above conflicts.

- **Conflict prevention**: A typical case in transactions is Read-modify-write, for which protocols like PAXOS or distributed locking provide a better solution. This is a common technique that can offer both arbitrary isolated transactions and atomic read-modify-write semantics. Yet another approach is to avoid concurrent writes
entirely and re-direct each writes of a particular data item to a one global node. There are many solutions available in literature to overcome the issue of database conflicts which are utilized in some systems with higher consistency guarantees. HBase, MongoDB and RDBMS use this approach to prevent conflicts.

- **Conflict detection:** A track of concurrent conflicting updates is kept in a database. It either rollback one of the inconsistent updates or it saves all versions of rollback to resolve on the user side. Concurrent updates are usually tracked by using vector clocks (which can be though as a common practice of the optimistic locking) or by saving an entire re-version history. This solution is used in systems like CouchDB, Riak, and Voldemort.

**Eventually Consistent Data Types:** In the above sections, we worked with the assumption of data versions of two nodes always merge. So, reconciliation of updates that are inconsistent is a tough task and is not so easy to make all replicas to join to a semantically true value. In Amazon Dynamo database the deleted items may resurface, which is a well-known example. Let us take a simple example that demonstrates this problem: Assuming a global counter and each database node can do either increment or decrement operation as maintained by the database. However, each node can preserve its own local counter as a single value which is scalar in nature, but these local counters cannot be used to merge by straight-forward addition/subtraction. Let’s take an example: there are three nodes A, B, and C and operation of type increment was applied thrice, once per node. If A fetches value from B and updates it to the local copy, C fetches from B, C fetches from A, then C ends up with value 4 which is incorrect. One solution to overcome these issues is use of a vector clock like data structure and preserve a pair of counters for each node:

```
class Counter {

  int[ ] plus

  int[ ] minus

  int NODE_ID
```

125
increment() { plus[NODE_ID]++ }

decrement() { minus[NODE_ID]++ }

get() { return sum(plus) – sum(minus) }

merge(Counter other) {
    for i in 1..MAX_ID {
        plus[i] = max(plus[i], other.plus[i])
        minus[i] = max(minus[i], other.minus[i])
    }
}

A likewise methodology is used by Cassandra to offer counters as a part of its features. It is easy to develop consistent data structures with increased complexity that can provide either state-based or operation-based replication strategies. For example such data structure includes:

1. Increment/decrement counters - counters
2. add-remove operations - sets
3. removeEdge / removeVertex / addEdge / addVertex operations – graph operations
4. insertAt(position) / removeAt(position) operations - lists

Although, data types generally impose inconsistent performance overheads and are limited in functionality.

6.3.3 Distributed Algorithms in Cloud Data Management

This section focuses on algorithms, which manage a distributed data store having inside data placement. Such algorithms are answerable for mapping among migration of data from one node to another, data items and physical nodes, and provisioning of global resources like RAM in the database.
**Data Placement – Rebalancing:** Here, we may start with a meek protocol which is focused to provide outage-free data migration amongst cluster nodes. This task arises in scenarios like cluster extension, failover, or act of rebalancing. Consider a scenario that is depicted in section (A) of the figure 6.15 – there are 3 nodes and all nodes contain a section of data that is distributed amongst the nodes according to a random data placement policy.

![Diagram A](image)

**Figure 6.15:** Data Rebalancing through Application of Data Migration Technique

If we don’t have a database, which offers internal data rebalancing, we will probably deploy multiple instances of the data store to every node as depicted in section (B) of the figure 6.15. It enables us to execute a cluster extension by turning a discrete instance off, copying it to another new node, and turning it on, as shown in section (C). Though an automatic database has the ability to track every record separately, many systems inclusive of *Oracle, MongoDB, Coherence, and Redis Cluster* utilize the described method internally, i.e. group records into shards. Obviously, number of shards should be quite enormous relative to the number of nodes to offer an even load distribution. An
outage-free shard migration can be achieved based on the simple protocol that routes client from the exporting to the importing node throughout the migration of shard.

### 6.3.4 Sharing and Replication in Dynamic Environments

The next query to be focused is to the mapping of records to the physical nodes. A simple solution is to have a table of key ranges with each range being assigned to a node or to use functions like $NodeID = \text{hash(key)} \% \text{TotalNodes}$. However, modulus-based hashing does not explicitly handle cluster reconfiguration as the removal or addition of nodes leads to complete reshuffling of data throughout the cluster. Because of this, it is difficult to handle replication and failover.

### SUMMARY

This chapter has determined on the storage aspect of cloud computing systems, precisely, NoSQL data stores and these technologies are serving as alternatives to the traditional DBMS. They are capable of managing huge volumes of data on cloud. Precisely the complete NoSQL data stores are reviewed in this chapter by providing academicians, researchers and practitioners against their usage, issues and challenges of using such technologies in real business scenarios. It is assessed among all the available NoSQL solutions on variety of dimensions including querying capacity of such data models, scalability issues, security threats and use cases. It also judges that the choice varies upon the type of applications. In addition, the work has verified challenges in the domain, including inconsistency, sparse comparison, terminology diversity, limited documentation, occasional immaturity of solutions, benchmarking criteria, and lack of support and non-existence of a standardized query language.