# Module - V

## System Architecture, Object Oriented Databases, XML and NoSQL

### Q1. What is the difference between Homogeneous and Heterogeneous Distributed DBMSs.

A **Distributed Database Management System (DDBMS)** may be classified as homogeneous or heterogeneous. In a **homogeneous distributed database system**, all sites have identical database management system software, In such a system, local sites surrender a portion of their autonomy in terms of their right to change schemas or database-management system software. That software must also cooperate with other sites in exchanging information about transactions, to make transaction processing possible across multiple sites.

In contrast, in a **heterogeneous distributed database**, different sites may use different schemas, and different database-management system software. The sites may not be aware of one another, and they may provide only limited facilities for cooperation in transaction processing. The differences in schemas are often a major problem for query processing, while the divergence in software becomes a hindrance for processing transactions that access multiple sites.

### Q2. What do you mean by a Distributed Data Storage? Point out two approaches in distributed data storage.

A distributed data store is a system that stores and processes data on multiple machines. Consider a relation $r$ that is to be stored in the database. There are two approaches to storing this relation in the distributed database:

**Replication.** The system maintains several identical replicas (copies) of the relation, and stores each replica at a different site.

**Fragmentation.** The system partitions the relation into several fragments, and stores each fragment at a different site.

#### Data Replication

If relation $r$ is replicated, a copy of relation $r$ is stored in two or more sites. If a copy is stored in every site in the system, it is known as **full replication**. There are a number of advantages and disadvantages to replication.

**Availability.** If one of the sites containing the relation $r$ fails, then the relation $r$ can be found in another site. Thus, the system can continue to process queries involving $r$, despite the failure of one site.

**Increased parallelism.** In the case where the majority of accesses to the relation $r$ result in only the reading of the relation, then several sites can process queries involving $r$ in parallel.

**Increased overhead on update.** The system must ensure that all replicas of a relation $r$ are consistent; otherwise, erroneous computations may result. Thus, whenever $r$ is updated, the update must be propagated to all sites containing replicas. This end up with an increased overhead.

### Data Fragmentation

If relation $r$ is fragmented, $r$ is divided into a number of fragments $r_1, r_2, ..., r_n$. These fragments contain sufficient information to allow reconstruction of the original relation $r$. There are two different schemes for fragmenting a relation: *horizontal fragmentation* and *vertical fragmentation*. Horizontal fragmentation splits the relation by assigning each tuple of $r$ to one or more fragments. Vertical fragmentation splits the relation by decomposing the scheme $R$ of relation $r$.

In **horizontal fragmentation**, a relation $r$ is partitioned into a number of subsets, $r_1, r_2, ..., r_n$. Each tuple of relation $r$ must belong to at least one of the fragments, so that the original relation can be reconstructed, if needed. In general, a horizontal fragment can be defined as a *selection* on the global relation $r$. That is, it use a predicate $P_i$ to construct fragment $r_i$ :

$$r_i = \sigma_{p_i}(r)$$

The relation $r$ can be reconstructed by taking the union of all fragments; that is:

$$r = r_1 \cup r_2 \cup ... \cup r_n$$

In its simplest form, vertical fragmentation is the same as decomposition. **vertical fragmentation** of $r(R)$ involves the definition of several subsets of attributes $R_1, R_2, ..., R_n$ of the schema $R$ so that:

$$R = R_1 \cup R_2 \cup ... \cup R_n$$

Each fragment $r_i$ of $r$ is defined by:

$$r = \pi_{R_i}(r)$$

The fragmentation should be done in such a way that we can reconstruct relation $r$ from the fragments by taking the natural join:

$$r = r_1 \bowtie r_2 \bowtie r_3 \bowtie ... \bowtie r_n$$

One way of ensuring that the relation $r$ can be reconstructed is to include the primary-key attributes of $R$ in each $R_i$.

## Q3. Explain the term data transparency in distributed database.

The user of a distributed database system should not be required to know where the data are physically located nor how the data can be accessed at the specific local site. This characteristic, called data transparency, can take several forms:

**Fragmentation transparency.** Users are not required to know how a relation has been fragmented.

**Replication transparency.** Users view each data object as logically unique. The distributed system may replicate an object to increase either system performance or data availability. Users do not have to be concerned with what data objects have been replicated, or where replicas have been placed.

**Location transparency.** Users are not required to know the physical location of the data. The distributed database system should be able to find any data as long as the data identifier is supplied by the user transaction.

### Q4. Differentiate between local transaction and global transaction.

In a distributed database system, the database is stored on several computers. The computers in a distributed system communicate with one another through various communication media, such as high-speed private networks or the Internet. The main differences between traditional databases and distributed databases are that distributed databases are typically geographically separated, are separately administered, and have a slower interconnection. Another major difference is that a transaction is differentiated as local and global transactions. A **local transaction** is one that accesses data only from sites where the transaction was initiated. A **global transaction**, on the other hand, is one that either accesses data in a site different from the one at which the transaction was initiated, or accesses data in several different sites.

### Q5. Explain the reason for building a distributed database system.

**Sharing data.** The major advantage in building a distributed database system is the provision of an environment where users at one site may be able to access the data residing at other sites. For instance, in a distributed university system, where each campus stores data related to that campus, it is possible for a user in one campus to access data in another campus.

**Autonomy.** The primary advantage of sharing data by means of data distribution is that each site is able to retain a degree of control over data that are stored locally. In contrast to a centralized system, where a database administrator of the the central site controls the database, responsibilities are delegated to the local database administrator of each site in distributed system. Depending on the design of the distributed database system, each administrator may have a different degree of *local autonomy*.

**Availability.** If one site fails in a distributed system, the remaining sites may be able to continue operating. In particular, if data items are replicated in several sites, a transaction needing a particular data item may find that item in any of several sites. Thus, the failure of a site does not necessarily imply the shutdown of the system.

## Object Oriented Databases

### Q6. Explain the need of object oriented database.

Traditional database applications have conceptually simple data types. The basic data items are records that are fairly small and whose fields are atomic — that is, they are not further structured.

In recent years, demand has grown for ways to deal with more complex data types. Consider, for example, *addresses*. While an entire address could be viewed as an atomic data item of type string, this view would hide details such as the street address, city, state, and postal code, which

could be of interest to queries. On the other hand, if an address were represented by breaking it into the components (street address, city, state, and postal code), writing queries would be more complicated since they would have to mention each field. A better alternative is to allow structured data types that allow a type *address* with subparts *street_address*, *city*, *state*, and *postal code*.

### Q7. What are structured datatypes in SQL?

Structured types allow composite attributes of E-R designs to be represented directly. For example, the following structured type can be defined to represent a composite attribute *name* with component attribute *firstname* and *lastname*:

```
create type Name as
    (firstname varchar(20),
    lastname varchar(20));
```

Similarly, the following structured type can be used to represent a composite attribute *address*:

```
create type Address as
    (street varchar(20),
    city varchar(20),
    zipcode varchar(9));
```

Such types are called **user-defined types** in SQL. Now these types can be used to create composite attributes in a relation, by simply declaring an attribute to be of one of these types. For example, a table *person* can be defined as follows:

```
create table person (
    name Name,
    address Address,
    dateOfBirth date);
```

The components of a composite attribute can be accessed using a "dot" notation; for instance *name.firstname* returns the firstname component of the name attribute. An access to attribute *name* would return a value of the structured type *Name*.

A structured type can have **methods** defined on it. The methods can be declared as part of the type definition of a structured type:

```
create type PersonType as (
    name Name,
    address Address,
    dateOfBirth date)
method ageOnDate(onDate date)
    returns interval year;
```

We create the method body separately:

```
create instance method ageOnDate (onDate date)
    returns interval year
    for PersonType
begin
    return onDate - self.dateOfBirth;
end
```

Note that the *for* clause indicates which type this method is for, while the keyword *instance* indicates that thi method executes on an instance of the *Person* type. The variable *self* refers to the *Person* instance on which the method is invoked.

The **constructor functions** are used to create values of structured types. A function with the same name as a structured type is a constructor function for the structured type. For instance, we could declare a constructor for the type *Name* like this:

```
create function Name (firstname varchar(20), lastname varchar(20))
returns Name
begin
    set self.firstname = firstname;
    set self.lastname = lastname;
end
```

## Q8. What are type inheritance in SQL?

By using the type inheritance a structured type in SQL can be inherited to form other subtypes. Suppose there is a structured type like,

```
create type Name as
    (name varchar(20),
    address varchar(20));
```

and want to store extra information in the database about people who are students, and about people who are teachers. Since students and teachers are also people, inheritance can be used to define the student and teacher types in SQL:

```
create type Student
    under Person
    (degree varchar(20),
    department varchar(20));


create type Teacher
    under Person
    (salary integer,
    department varchar(20));
```

Both *Student* and *Teacher* inherit the attributes of *Person* - namely, *name* and *address*. *Student* and *Teacher* are said to be subtypes of *Person*, and *Person* is a supertype of *Student*, as well as of *Teacher*.

The SQL standard requires an extra field at the end of the type definition, whose value is either *final* or *not final*. The keyword *final* says that subtypes may not be created from the given type, while *not final* says that subtypes may be created.

### Q9. Define XML. What are the advantageous of XML? Discuss XML Featues.

Companies are using the internet to create new types of systems that integrate their data to increase efficiency and reduce costs. Electronic commerce (e-commerce) enables all types of organizations to market and sell products and services to a global market of millions of users. E-commerce transactions—the sale of products or services—can take place between businesses (business-to-business, or B2B) or between a business and a consumer (business-to-consumer, or B2C).

However, if an application wants to get the order data from the Web page, there is no easy way to extract the order details (such as the order number, the date, the customer number, the item, the quantity, the price, or payment details) from an HTML document. The HTML document can only describe how to display the order in a Web browser; it does not permit the manipulation of the order's data elements, that is, date, shipping information, payment details, product information, and so on. To solve that problem, a new markup language, known as Extensible Markup Language, or XML, was developed.

**Extensible Markup Language (XML)** is a meta-language used to represent and manipulate data elements. XML is designed to facilitate the exchange of structured documents, such as orders and invoices, over the Internet. The XML metalanguage allows the definition of news, such as $< ProdPrice >$, to describe the data elements used in an XML document. This ability to extend the language explains the $X$ in XML; the language is said to be extensible. XML is derived from the Standard Generalized Markup Language (SGML), an international standard for the publication and distribution of highly complex technical documents. XML has very important characteristics, as follows:

- XML allows the definition of new tags to describe data elements, such as $< ProductId >$.
- XML is case sensitive: $< ProductID >$ is not the same as $< Productid >$.
- XMLs must be well formed; that is, tags must be properly formatted. Most openings also have a corresponding closing. For example, the product identification would require the format $< ProductId > 2345 - AA < /ProductId >$.
- XMLs must be properly nested. For example, a properly nested XML might look like this: $< Product >< ProductId > 2345 - AA < /ProductId >< /Product >$.
- It use the $< --$ and $-- >$ symbols to enter comments in the XML document.
- The XML and xml prefixes are reserved for XMLs only.

As an illustration of the use of XML for data exchange purposes, consider a B2B example in which Company A uses XML to exchange product data with Company B over the internet. Figure 38 shows the contents of the *ProductList.xml* document.

The XML example shown in Figure 38 illustrates several important XML features, as follows:

- The first line represents the XML document declaration, and it is mandatory.
- Every XML document has a root element. In the example, the second line declares the *ProductList* root element.
- The root element contains child elements or subelements. In the example, line 3 declares *Product* as a child element of *ProductList*.

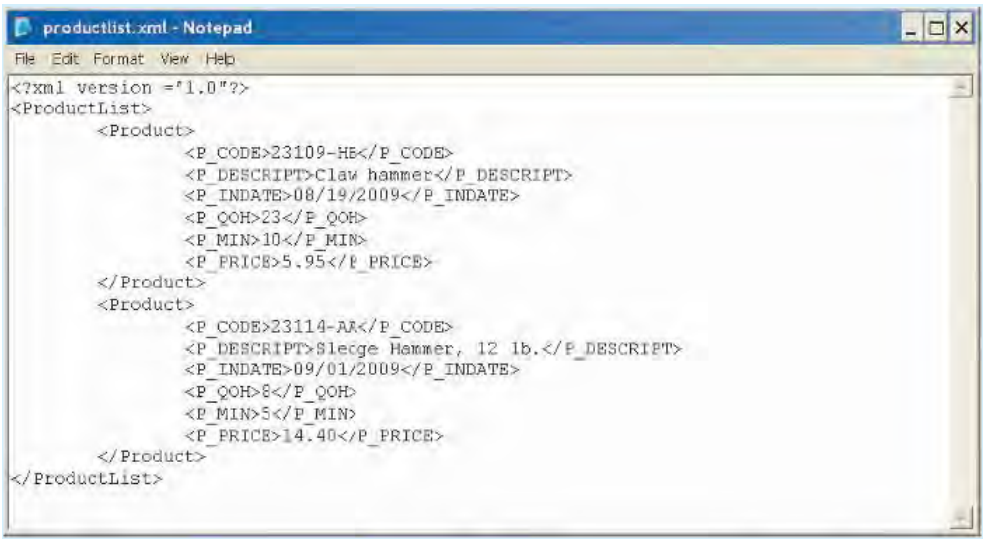


Figure 38: Contents of the productlist.xml document.

- Each element can contain subelements. For example, each *Product* element is composed of several child elements, represented by *P_CODE*, *P_DESCRIPT*, *P_INDATE*, *P_QOH*, *P_MIN*, and *P_PRICE*.
- The XML document reflects a hierarchical tree structure where elements are related in a parent-child relationship; each parent element can have many child elements.

## Q10. Write short notes on Document Type Definition and XML Schemas.

### Document Type Definition

Companies that use B2B transactions must have a way to understand and validate each other's tags. One way to accomplish that task is through the use of Document Type Definitions. A **Document Type Definition (DTD)** is a file with a .dtd extension that describes XML elements—in effect, a DTD file provides the composition of the database's logical model and defines the syntax rules or valid elements for each type of XML document. Figure 39 shows the *productlist.dtd* document for the *productlist.xml* document shown earlier in Figure 38.

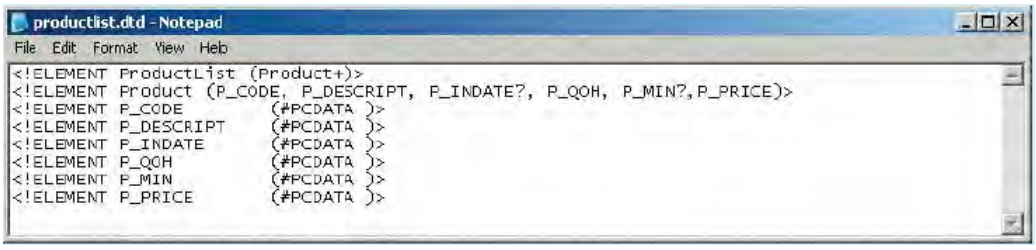


Figure 39: Contents of the productlist.dtd document.

In Figure 39, note that the *productlist.dtd* file provides definitions of the elements in the *productlist.xml* document. In particular, note that:

- The first line declares the *ProductList* root element.

- The *ProductList* root element has one child, the Product element.
- The plus "+" symbol indicates that *Product* occurs one or more times within *ProductList*.
- An asterisk "*" would mean that the child element occurs zero or more times.
- A question mark "?" would mean that the child element is optional.
- The second line describes the *Product* element.
- The question mark "?" after the *P_INDATE* and *P_MIN* indicates that they are optional elements.
- The third through eighth lines show that the *Product* element has six child elements.
- The *#PCDATA* keyword represents the actual text data.

### XML Schema

The **XML schema** is an advanced data definition language that is used to describe the structure (elements, data types, relationship types, ranges, and default values) of XML data documents. One of the main advantages of an XML schema is that it more closely maps to database terminology and features. For example, an XML schema will be able to define common database types such as date, integer, or decimal; minimum and maximum values; a list of valid values; and required elements. Using the XML schema, a company would be able to validate the data for values that may be out of range, incorrect dates, valid values, and so on.

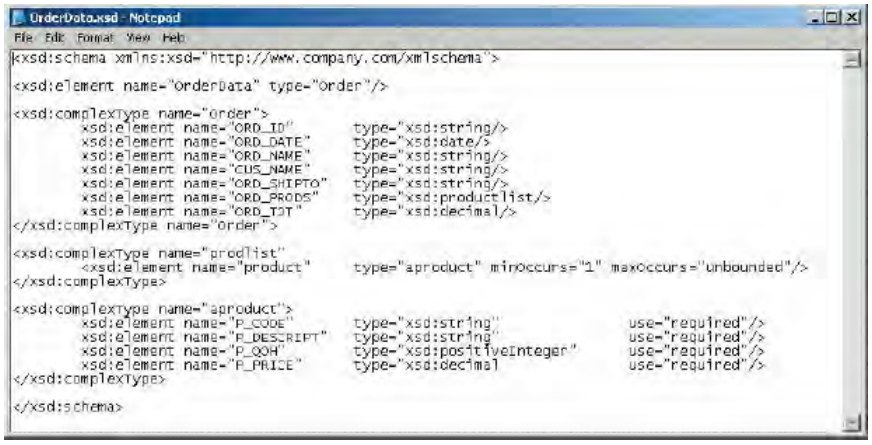


Figure 40: DTD and XML documents for order data.

Unlike a DTD document, which uses a unique syntax, an **XML schema definition (XSD)** file uses a syntax that resembles an XML document. Figure 40 shows the XSD document for the *OrderData* XML document.

## Next Generation Databases

### Q11. How data replication is achieved in distributed relational database.

Database replication adopted as a means of achieving high availability. Using replication, database administrators could configure a standby database that could take over for the primary database

in the event of failure. Database replication often took advantage of the transaction log that most relational databases used to support ACID transactions. When a transaction commits in an ACID-compliant database, the transaction record is immediately written to the transaction log so that it is preserved in the event of failure. A replication process monitoring the transaction log can apply changes to a backup database, thereby creating a replica.

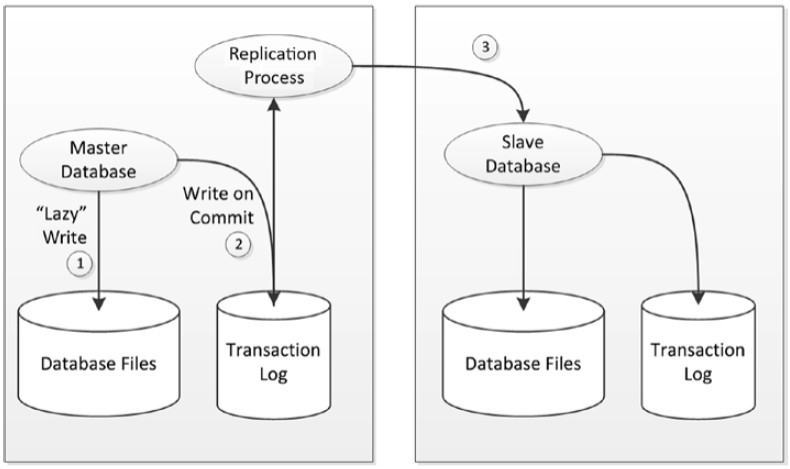


Figure 41: Log-based replication.

Figure 41 illustrates the log-based replication approach. Database transactions are written in an asynchronous "lazy" manner to the database files (1), but a database transaction immediately writes to the transaction log upon commit (2). The replication process monitors the transaction log and applies transactions as they are written to the read-only slave database (3). Replication is usually asynchronous, but in some databases the commit can be deferred until the transaction has been replicated to the slave.

## Q12. Explain how sharding and replication is implemented in MongoDB.

MongoDB supports sharding to provide scale-out capabilities and replication for high availability.

### Sharding

A high-level representation of the MongoDB sharding architecture is shown in Figure 42. Each shard is implemented by a distinct MongoDB database, which in most respects is unaware of its role in the broader sharded server (1). A separate MongoDB database—the config server (2)—contains the metadata that can be used to determine how data is distributed across shards. A router process (3) is responsible for routing requests to the appropriate shard server.

### Sharding Mechanisms

Distribution of data across shards can be either *range based* or *hash based*. In range-based partitioning, each shard is allocated a specific range of shard key values. MongoDB consults the distribution of key values in the index to ensure that each shard is allocated approximately the same number of keys. In hash-based sharding, the keys are distributed based on a hash function applied to the shard key.

There are advantages and compromises involved in each scheme. Figure 43 illustrates the performance trade-offs inherent in range and hash sharding for inserts and range queries.

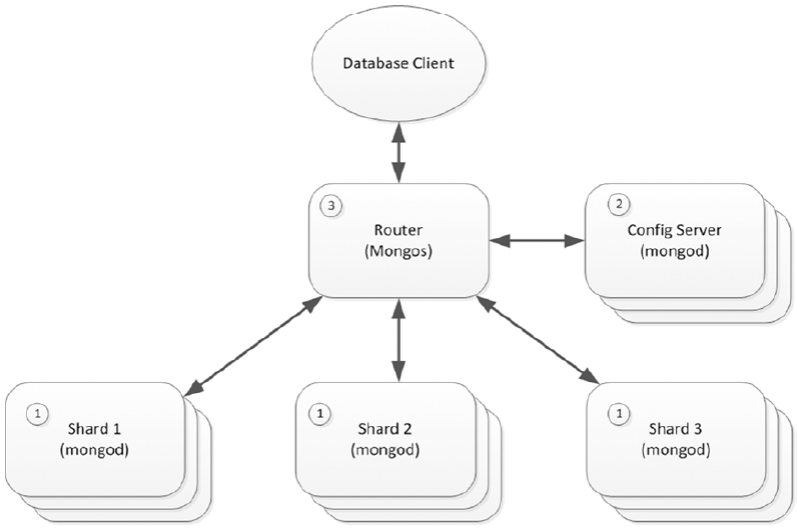


Figure 42: MongoDB sharding architecture.

Range-based partitioning allows for more efficient execution of queries that process ranges of values, since these queries can often be resolved by accessing a single shard. Hash-based sharding requires that range queries be resolved by accessing all shards. On the other hand, hash-based sharding is more likely to distribute "hot" documents evenly across the cluster, thus balancing the load more effectively.

However, when range partitioning is enabled and the shard key is continuously incrementing, the load tends to aggregate against only one of the shards, thus unbalancing the cluster. With hash-based partitioning new documents are distributed evenly across all members of the cluster. Furthermore, although MongoDB tries to distribute shard keys evenly across the cluster, it may be that there are hotspots within particular shard key ranges which again unbalance the load. Hash-based sharding is more likely to evenly distribute the load in this scenario.

### Replication

In MongoDB, data can be replicated across machines by the means of replica sets. A replica set consists of a primary node together with two or more secondary nodes. The primary node accepts all write requests, which are propagated asynchronously to the secondary nodes.

The primary node is determined by an election involving all available nodes. To be eligible to become primary, a node must be able to contact more than half of the replica set. This ensures that if a network partitions a replica set in two, only one of the partitions will attempt to establish a primary.

The successful primary will be elected based on the number of nodes to which it is in contact, together with a priority value that may be assigned by the system administrator. Setting a priority of 0 to an instance prevents it from ever being elected as primary. In the event of a tie, the server with the most recent *optime*-the timestamp of the last operation-will be selected.

Members within a replica set communicate frequently via heartbeat messages. If a primary finds it is unable to receive heartbeat messages from more than half of the secondaries, then it will renounce its primary status and a new election will be called. Figure 44 illustrates a three-member replica set and shows how a network partition leads to a change of primary.

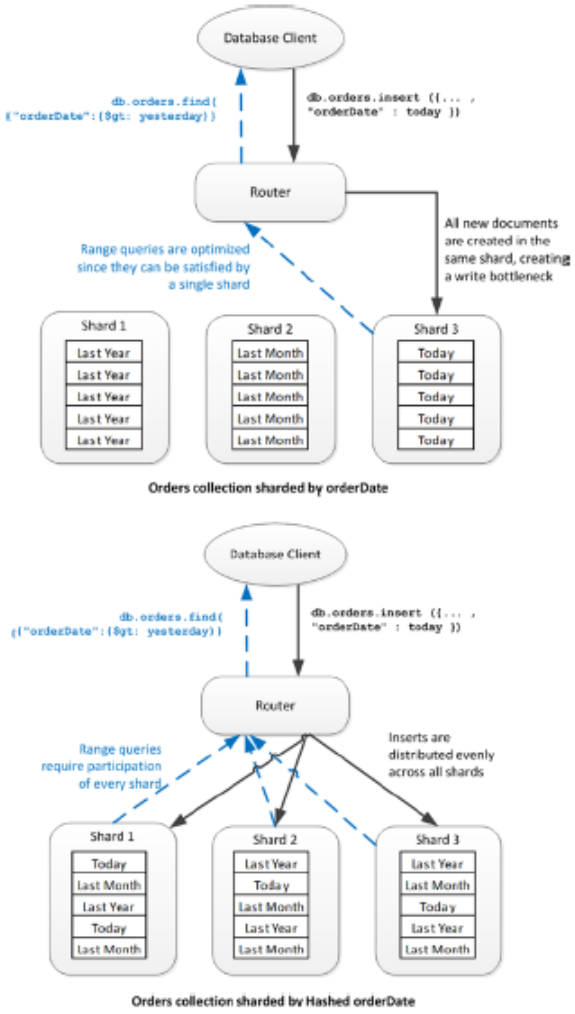


Figure 43: Comparison of range and hash sharding in MongoDB.

## Q13. Write short notes on Hbase.

HBase is a mechanism for providing random access database services on top of the Hadoop HDFS file system, or HBase can be thought of as an open-source implementation of Google's BigTable database that happens to use HDFS for data storage. On the other hand, HBase implements real-time random access database functionality, which is essentially distinct from the base capabilities of Hadoop.

The implementation of HBase over HDFS creates a sort of hybrid, a mix of shared-nothing and shared-disk clustering patterns. On the one hand, every HBase node can access any element of data in the database because all data is accessible via HDFS. On the other hand, it is typical to co-locate HBase servers with HDFS DataNodes, which means that in practice each node tends to be responsible for an exclusive subset of data stored on local disk. In either case, HDFS provides the reliability guarantees for data on disk: the HBase architecture is not required to concern itself with write mirroring or disk failure management, because these are handled automatically by the underlying HDFS system.

HBase implements a wide column store based on Google's BigTable specification. All rows in an HBase table are identified by a unique *row key*. A table of nontrivial size will be split into

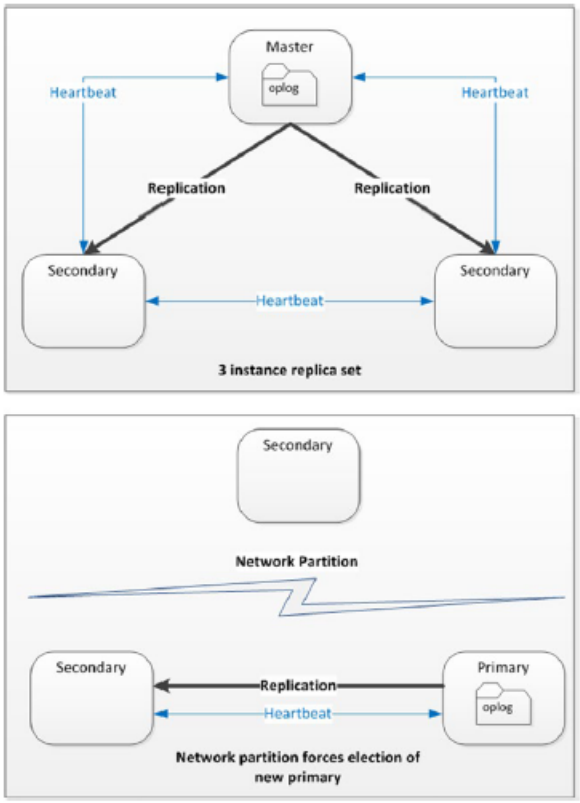


Figure 44: MongoDB replica set and primary failover.

multiple horizontal partitions called *regions*. Each region consists of a contiguous, sorted range of key values. Read or write access to a region is controlled by a *RegionServer*. Each RegionServer normally runs on a dedicated host, and is typically co-located with the Hadoop DataNode. There will usually be more than one region in each RegionServer. As regions grow, they split into multiple regions based on configurable policies. Regions may also be split manually.

Each HBase installation will include a Hadoop *Zookeeper* service that is implemented across multiple nodes. Hbase may share this Zookeeper ensemble with the rest of the Hadoop cluster or use a dedicated service. The HBase *master server* performs a variety of housekeeping tasks. In particular, it controls the balancing of regions among RegionServers. If a RegionServer is added or removed, the master will organize for its regions to be relocated to other RegionServers.

Figure 45 illustrates some of these architectural elements. An HBase client consults Zookeeper to determine the location of the HBase catalog tables (1), which can be then be interrogated to determine the location of the appropriate RegionServer (2). The client will then request to read or modify a key value from the appropriate RegionServer (3). The RegionServer reads or writes to the appropriate disk files, which are located on HDFS (4).

### Q14. Give an illustration of data locality in Hbase.

All rows in an HBase table are identified by a unique *row key*. A table of nontrivial size will be split into multiple horizontal partitions called *regions*. Each region consists of a contiguous, sorted range of key values. Read or write access to a region is controlled by a *RegionServer*.

The RegionServer includes a *block cache* that can satisfy many reads from memory, and a

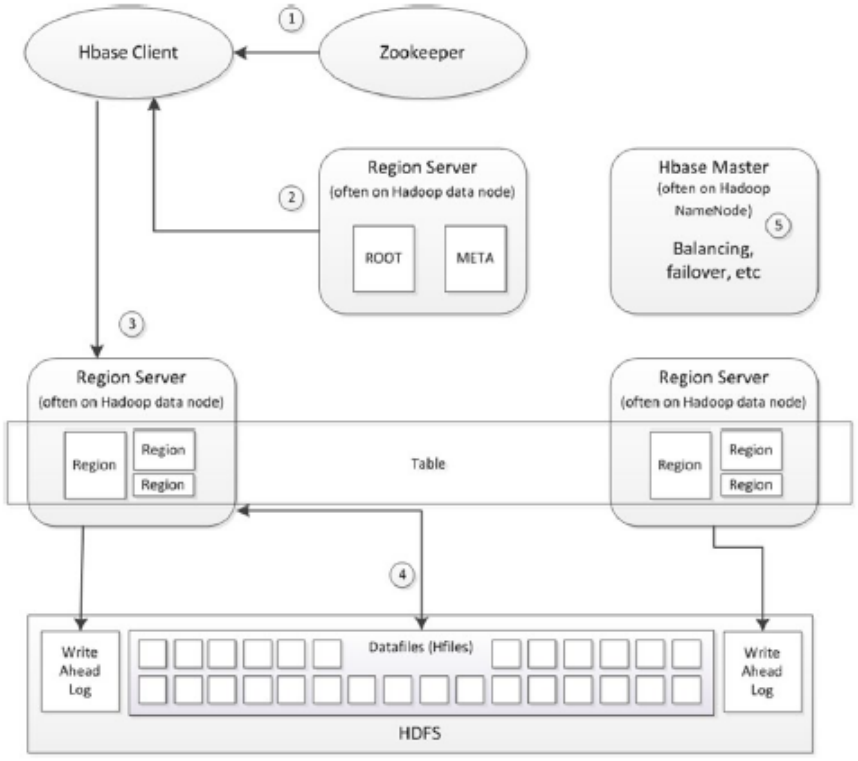


Figure 45: HBase architecture.

*MemStore*, which writes in memory before being flushed to disk. However, to ensure durability of the writes, each RegionServer has a dedicated *write ahead log (WAL)*, which journals all writes to HDFS. Each RegionServer is located on a Hadoop NameNode, and as a result, region data will be co-located with the RegionServer, providing good *data locality*.

The three levels of data locality are shown in Figure 46. In the first configuration, the RegionServer and the DataNode are located on different servers and all reads and writes have to pass across the network. In the second configuration, the RegionServer and the DataNode are co-located and all reads and writes pass through the DataNode, but they are satisfied by the local disk. In the third scenario, short-circuit reads are configured and the RegionServer can read directly from the local disk.

## Q15. Write short notes on Cassandra.

In Cassandra, there are no specialized master nodes. Every node is equal and every node is capable of performing any of the activities required for cluster operation. Nodes in Cassandra do, however, have short-term specialized responsibilities. For instance, when a client performs an operation, a node will be allocated as the *coordinator* for that operation. When a new member is added to the cluster, a node will be nominated as the *seed node* from which the new node will seek information. However, these short-term responsibilities can be performed by any node in the cluster.

One of the advantages of a master node is that it can maintain a canonical version of cluster configuration and state. In the absence of such a master node, Cassandra requires that all members of the cluster be kept up to date with the current state of cluster configuration and status. This is achieved by use of the **gossip** protocol. Every second each member of the cluster will transmit

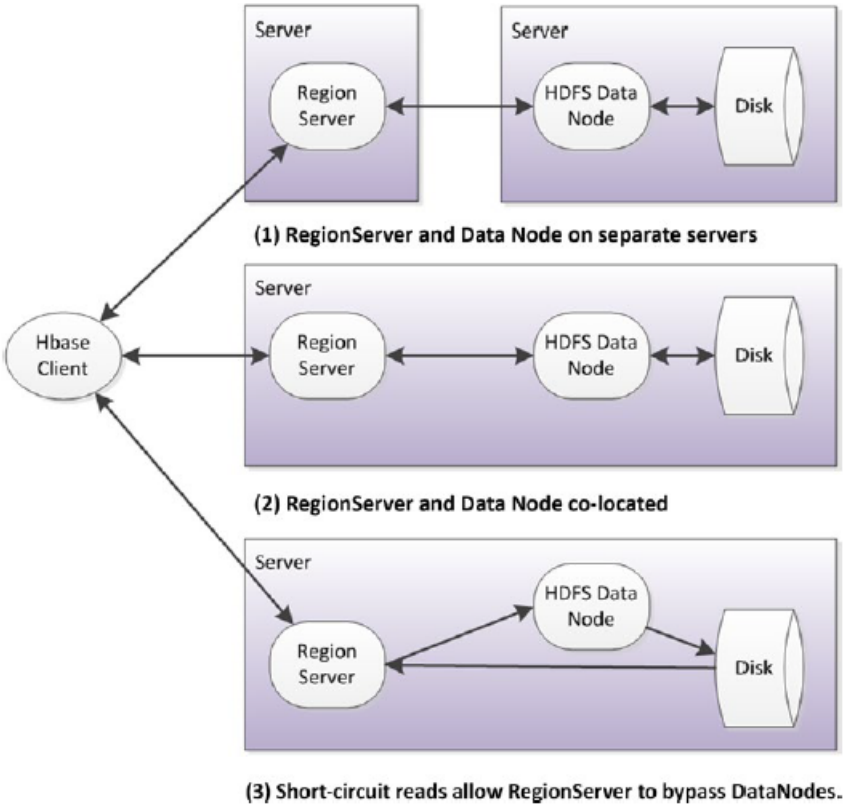


Figure 46: Data locality in HBase.

information about its state and the state of any other nodes it is aware of to up to three other nodes in the cluster. In this way, cluster status is constantly being updated across all members of the cluster. The gossip protocol, by its literal names suggests, in Cassandra, the nodes gossip about other nodes as well as about their own state.

Cluster configuration is persisted in the system *keyspace*, which is available to all members of the cluster. A keyspace is roughly analogous to a schema in a relational database—the system keyspace contains tables that record metadata about the cluster configuration.

### Q16. Explain consistent hashing algorithm in Cassandra.

Cassandra databases distribute data throughout the cluster by using consistent hashing. The rowkey (analogous to a primary key in an RDBMS) is hashed. Each node is allocated a range of hash values, and the node that has the specific range for a hashed key value takes responsibility for the initial placement of that data.

In the default Cassandra partitioning scheme, the hash values range from $-2^{63}$ to $2^{63} - 1$. Therefore, if there were four nodes in the cluster and we wanted to assign equal numbers of hashes to each node, then the hash ranges for each would be approximately as follows:

| Node | Low Hash | High Hash |
|---|---|---|
| Node A | $-2^{63}$ | $-2^{63}/2$ |
| Node B | $-2^{63}/2$ | 0 |
| Node C | 0 | $2^{63}/2$ |
| Node D | $2^{63}/2$ | $2^{63}$ |

Figure 47 illustrates simple consistent hashing: the value for a rowkey is hashed, which determines its position on "the ring". Nodes in the cluster take responsibility for ranges of values within the ring, and therefore take ownership of specific rowkey values.

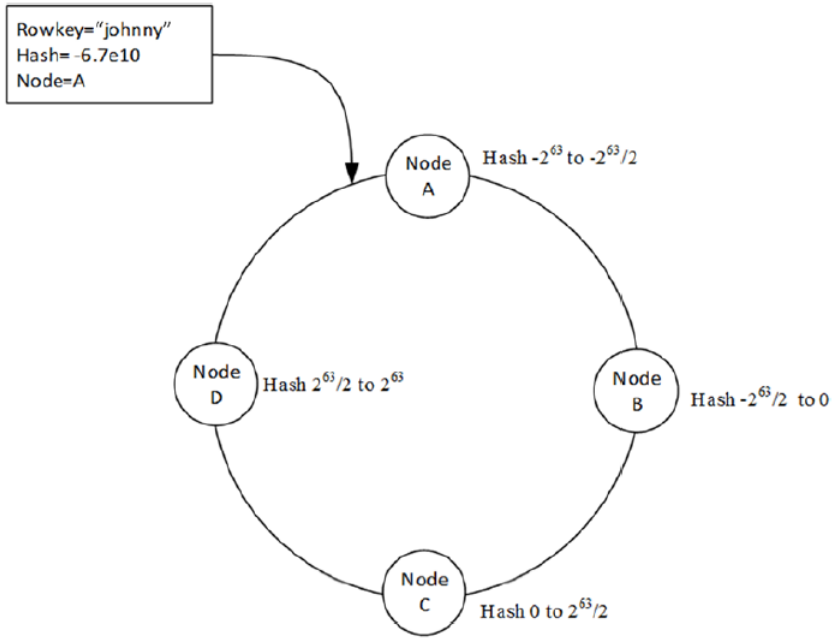


Figure 47: Consistent hashing.

The four-node cluster in Figure 47 is well balanced because every node is responsible for hash ranges of similar magnitude. But we risk unbalancing the cluster as we add nodes. If we double the number of nodes in the cluster, then we can assign the new nodes at points on the ring between existing nodes and the cluster will remain balanced.

Early versions of Cassandra had two options when adding a new node. We could either remap all the hash ranges, or we could map the new node within an existing range. In the first option we obtain a balanced cluster, but only after an expensive rebalancing process. In the second option the cluster becomes unbalanced; since each node is responsible for the region of the ring between itself and its predecessor, adding a new node without changing the ranges of other nodes essentially splits a region in half.

## Q17. Highlight CAP theorem and its importance.

In 2000, Eric Brewer outlined the "CAP" conjecture, which was later granted theorem status when a mathematical proof was provided. The CAP theorem says that in a distributed database system, it can have at most only two of Consistency, Availability, and Partition tolerance.

**Consistency** means that every user of the database has an identical view of the data at any given instant. **Availability** means that in the event of a failure, the database remains operational.

**Partition tolerance** means that the database can maintain operations in the event of the network's failing between two segments of the distributed system.

In normal operations, the data store provides all three functions. But the CAP theorem maintains that when a distributed database experiences a network failure, it can provide either consistency or availability. In the theorem, partition tolerance is a must. The assumption is that the system operates on a distributed data store so the system, by nature, operates with network partitions. Network failures will happen, so to offer any kind of reliable service, partition tolerance is necessary - the P of CAP.

That leaves a decision between the other two, C and A. When a network failure happens, one can choose to guarantee consistency or availability:

- High consistency comes at the cost of lower availability.
- High availability comes at the cost of lower consistency.