SAINTGITS
LEARN.GROW.EXCEL

# Module 5

SAINTGITS
LEARN.GROW.EXCEL

# Distributed Databases

- A distributed database system consists of loosely coupled sites that share no physical components.
- Each site may participate in the execution of transactions that access data at one site, or several sites.
- The main difference between centralized and distributed database systems is that, in the former, the data reside in one single location, whereas in the latter, the data reside in several locations.
- This distribution of data is the cause of many difficulties in transaction processing and query processing.

SAINTGITS
LEARN.GROW.EXCEL

# Homogeneous and Heterogeneous Databases

- In a **homogeneous distributed database system**, all sites have identical database management system software, are aware of one another, and agree to cooperate in processing users' requests.
- 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.
- 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.

SAINTGITS
LEARN.GROW.EXCEL

# Homogeneous and Heterogeneous Databases

- In a **homogeneous distributed database system**, all sites have identical database management system software, are aware of one another, and agree to cooperate in processing users' requests.
- 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.
- 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.

SAINTGITS
LEARN.GROW.EXCEL

# Distributed Data Storage

- 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. The alternative to replication is to store only one copy of relation r.
- **Fragmentation:** The system partitions the relation into several fragments, and stores each fragment at a different site.
- Fragmentation and replication can be combined.
- A relation can be partitioned into several fragments and there may be several replicas of each fragment.

# Data Replication

- If relation r is replicated, a copy of relation r is stored in two or more sites.
- In the most extreme case, we have full replication, in which a copy is stored in every site in the system.
- There are a number of advantages and disadvantages to replication.
- **Availability** : If one of the sites containing 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. The more replicas of r there are, the greater the chance that the needed data will be found in the site where the transaction is executing. Hence, data replication minimizes movement of data between sites.

# Data Replication

- **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. The result is increased overhead.
- For example, in a banking system, where account information is replicated in various sites, it is necessary to ensure that the balance in a particular account agrees in all sites.
- In general, replication enhances the performance of read operations and increases the availability of data to read-only transactions.
- However, update transactions incur greater overhead.
- Controlling concurrent updates by several transactions to replicated data is more complex than in centralized systems.
- We can simplify the management of replicas of relation r by choosing one of them as the primary copy of r.
- For example, in a banking system, an account can be associated with the site in which the account has been opened.
- Similarly, in an airline-reservation system, a flight can be associated with the site at which the flight originates.

SAINTGITS
LEARN.GROW.EXCEL

# Data Fragmentation

- If relation r is fragmented, r is divided into a number of fragments r1, r2, . . . , rn.
- 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, r1, r2, . . . , rn.
- Each tuple of relation r must belong to at least one of the fragments, so that the original relation can be reconstructed, if needed.

SAINTGITS
LEARN.GROW.EXCEL

# Data Fragmentation

- In horizontal fragmentation, a relation r is partitioned into a number of subsets, r1, r2, . . . , rn.
- Each tuple of relation r must belong to at least one of the fragments, so that the original relation can be reconstructed, if needed.
- As an illustration, the account relation can be divided into several different fragments, each of which consists of tuples of accounts belonging to a particular branch.
- If the banking system has only two branches—Hillside and Valleyview — then there are two different fragments:

$$account_1 = \sigma_{branch\_name = \text{“Hillside”}}(account)$$
$$account_2 = \sigma_{branch\_name = \text{“Valleyview”}}(account)$$

- In general, a horizontal fragment can be defined as a selection on the global relation r. That is,we use a predicate Pi to construct fragment ri :

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

SAINTGITS
LEARN.GROW.EXCEL

# Data Fragmentation

- We reconstruct the relation r by taking the union of all fragments; that is:

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

- **Vertical fragmentation** of r(R) involves the definition of several subsets of attributes R1, R2, . . . , Rn of the schema R so that:

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

- Each fragment ri of r is defined by

$$r_i = \Pi_{R_i}(r)$$

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

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

# Data Fragmentation

- To illustrate vertical fragmentation, consider a university database with a relation employee info that stores, for each employee, employee id, name, designation, and salary.
- For privacy reasons, this relation may be fragmented into a relation employee private info containing employee id and salary, and another relation employee public info containing attributes employee id, name, and designation.
- These may be stored at different sites, again, possibly for security reasons.

# Transparency

- 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 is called **data transparency**.
- 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.

SAINTGITS
LEARN.GROW.EXCEL

# Distributed Transactions

- Access to the various data items in a distributed system is usually accomplished through transactions, which must preserve the ACID properties
- There are two types of transaction that we need to consider.
- The local transactions are those that access and update data in only one local database; the global transactions are those that access and update data in several local databases.
- Ensuring the ACID properties for global transactions is much more complicated, since several sites may be participating in execution.
- The failure of one of these sites, or the failure of a communication link connecting these sites, may result in erroneous computations.

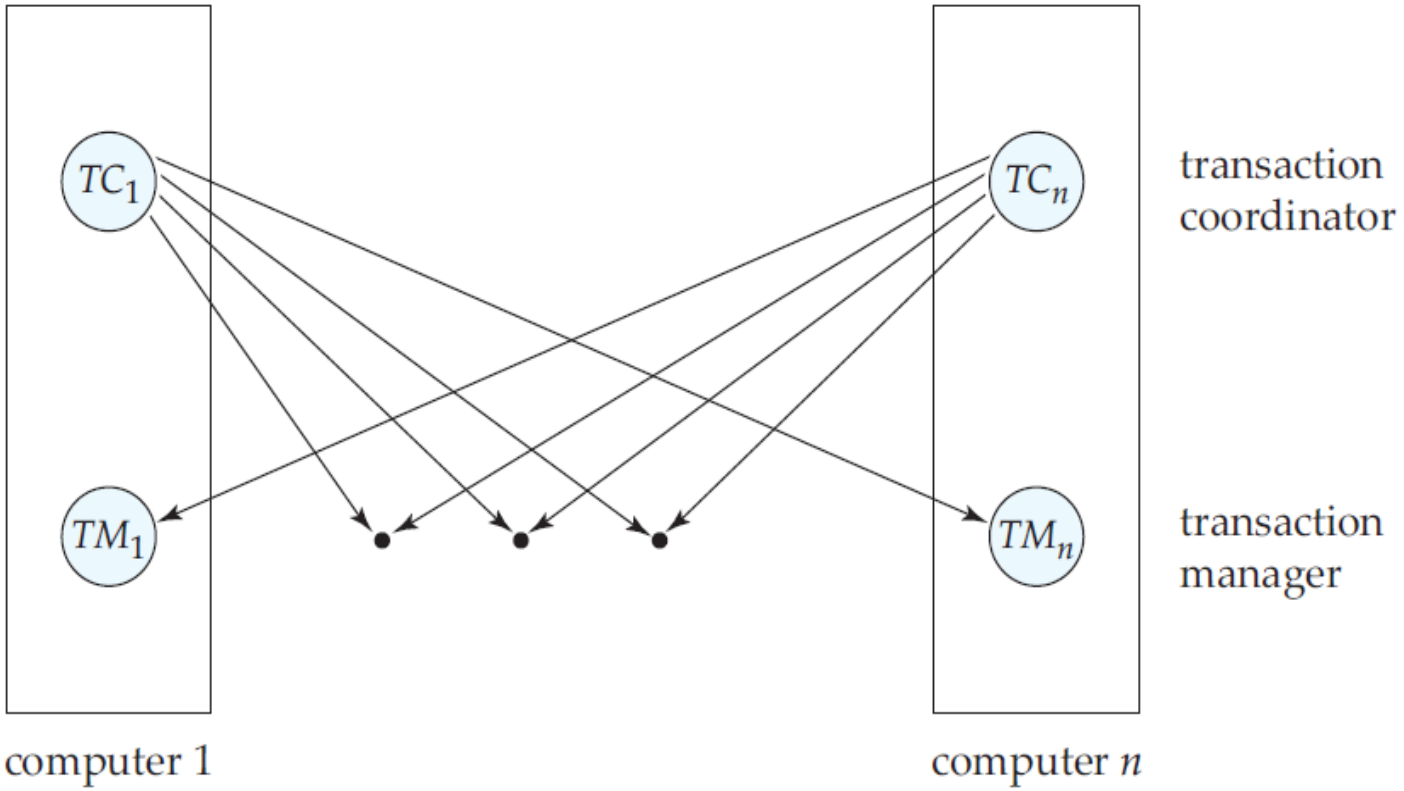
# System Structure

- Each site has its own local transaction manager, whose function is to ensure the ACID properties of those transactions that execute at that site.
- The various transaction managers cooperate to execute global transactions.
- The **transaction manager** manages the execution of those transactions (or subtransactions) that access data stored in a local site. Note that each such transaction may be either a local transaction (that is, a transaction that executes at only that site) or part of a global transaction (that is, a transaction that executes at several sites).
- The **transaction coordinator** coordinates the execution of the various transactions (both local and global) initiated at that site.
- The structure of a transaction manager is similar in many respects to the structure of a centralized system.
- Each transaction manager is responsible for:
  - Maintaining a log for recovery purposes.
  - Participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that site.

# System Structure

- The transaction coordinator subsystem is not needed in the centralized environment, since a transaction accesses data at only a single site.
- A transaction coordinator, as its name implies, is responsible for coordinating the execution of all the transactions initiated at that site.
- For each such transaction, the coordinator is responsible for:
  - Starting the execution of the transaction.
  - Breaking the transaction into a number of subtransactions and distributing these subtransactions to the appropriate sites for execution.
  - Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

# System Structure




System architecture.

# System Failure Modes

- A distributed system may suffer from the same types of failure that a centralized system does (for example, software errors, hardware errors, or disk crashes).
- There are, however, additional types of failure with which we need to deal in a distributed environment.
- The basic failure types are:
  - Failure of a site.
  - Loss of messages.
  - Failure of a communication link.
  - Network partition.
- The loss or corruption of messages is always a possibility in a distributed system.
- The system uses transmission-control protocols, such as TCP/IP, to handle such errors.

# System Failure Modes

- However, if two sites A and B are not directly connected, messages from one to the other must be routed through a sequence of communication links.
- If a communication link fails, messages that would have been transmitted across the link must be rerouted.
- In some cases, it is possible to find another route through the network, so that the messages are able to reach their destination.
- In other cases, a failure may result in there being no connection between some pairs of sites.
- A system is partitioned if it has been split into two (or more) subsystems, called partitions, that lack any connection between them.
- Note that, under this definition, a partition may consist of a single node.

SAINTGITS
LEARN.GROW.EXCEL

# Object Based Databases

- The first obstacle faced by programmers using the relational data model was the limited type system supported by the relational model.
- Complex application domains require correspondingly complex data types, such as nested record structures,multivalued attributes, and inheritance,which are supported by traditional programming languages.
- Such features are in fact supported in the E-R and extended E-R notations, but had to be translated to simpler SQL data types.
- The object-relational data model extends the relational data model by providing a richer type system including complex data types and object orientation.
- Relational query languages, in particular SQL, need to be correspondingly extended to deal with the richer type system.
- Object-relational database systems, that is, database systems based on the object-relationmodel, provide a convenient migration path for users of relational databases who wish to use object-oriented features.

SAINTGITS
LEARN.GROW.EXCEL

# Object Based Databases

- The second obstacle was the difficulty in accessing database data from programs written in programming languages such as C++ or Java.
- Merely extending the type system supported by the database was not enough to solve this problem completely.
- Differences between the type system of the database and the type system of the programming language make data storage and retrieval more complicated, and need to be minimized.
- We then address the issue of supporting persistence for data that is in the
- native type system of an object-oriented programming language. Two approaches are used in practice:

1. Build an **object-oriented database system**, that is, a database system that natively supports an object-oriented type system, and allows direct access to data from an object-oriented programming language using the native type system of the language.
2. Automatically convert data from the native type system of the programming language to a relational representation, and vice versa. Data conversion is specified using an **object-relational mapping**.

SAINTGITS
LEARN.GROW.EXCEL

# Complex Data Types

- Traditional database applications have conceptually simple data types.
- The basic data items are records that are fairly small and whose fields are atomic.
- 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.

# Complex Data Types

- Consider, for example, a library application, and suppose we wish to store the following information for each book:
  - Book title.
  - List of authors.
  - Publisher.
  - Set of keywords.
- We can see that, if we define a relation for the preceding information, several domains will be non atomic.
- **Authors -** A book may have a list of authors, which we can represent as an array. Nevertheless, we may want to find all books of which Jones was one of the authors. Thus, we are interested in a subpart of the domain element “authors.”
- **Keywords -** If we store a set of keywords for a book, we expect to be able to retrieve all books whose keywords include one or more specified keywords. Thus, we view the domain of the set of keywords as nonatomic.

# Complex Data Types

- **Publisher** - Unlike keywords and authors, publisher does not have a set-valued domain. However, we may view publisher as consisting of the subfields name and branch. This view makes the domain of publisher nonatomic.

| *title* | *author_array* | *publisher* | *keyword_set* |
|---|---|---|---|
| | | *(name, branch)* | |
| Compilers | [Smith, Jones] | (McGraw-Hill, NewYork) | {parsing, analysis} |
| Networks | [Jones, Frick] | (Oxford, London) | {Internet, Web} |

Non-1NF books relation, *books*.

- For simplicity, we assume that the title of a book uniquely identifies the book.
- We can then represent the same information using the following schema, where the primary key attributes are underlined.
  - *authors*(<u>*title*</u>, *author*, *position*)
  - *keywords*(*title*, *keyword*)
  - *books4*(<u>*title*</u>, *pub_name*, *pub_branch*)

SAINTGITS
LEARN.GROW.EXCEL

# Complex Data Types

| *title* | *author* | *position* |
|---|---|---|
| Compilers | Smith | 1 |
| Compilers | Jones | 2 |
| Networks | Jones | 1 |
| Networks | Frick | 2 |

*authors*

| *title* | *keyword* |
|---|---|
| Compilers | parsing |
| Compilers | analysis |
| Networks | Internet |
| Networks | Web |

*keywords*

| *title* | *pub_name* | *pub_branch* |
|---|---|---|
| Compilers | McGraw-Hill | New York |
| Networks | Oxford | London |

*books4*

4NF version of the relation *books*.

# Structured Types

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

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

- 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))
        not final;
```

# Structured Types

- Such types are called user-defined types in SQL.
- The final and not final specifications are related to subtyping.
- We can now use these types to create composite attributes in a relation, by simply declaring an attribute to be of one of these types.
- For example, we could create a table person 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.

SAINTGITS
LEARN.GROW.EXCEL

# Structured Types

- We can also create a table whose rows are of a user-defined type.
- For example, we could define a type PersonType and create the table person as follows.

```
create type PersonType as (
        name Name,
        address Address,
        dateOfBirth date)
        not final
create table person of PersonType;
```

- An alternativeway of defining composite attributes in SQL is to use unnamed row types.
- For instance, the relation representing person information could have been created using row types as follows.

```
create table person_r (
        name row (firstname varchar(20),
                  lastname varchar(20)),
        address row (street varchar(20),
                     city varchar(20),
                     zipcode varchar(9)),
        dateOfBirth date);
```

# Structured Types

- The following query illustrates how to access component attributes of a composite attribute. The query finds the last name and city of each person.

```
select name.lastname, address.city
from person;
```

- A structured type can have methods defined on it. We declare methods as part of the type definition of a structured type.

```
create type PersonType as (
        name Name,
        address Address,
        dateOfBirth date)
        not final
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
```

SAINTGITS
LEARN.GROW.EXCEL

# Structured Types

- The following statement illustrates how we can create a new tuple in the Person relation.

```
insert into Person
        values
                (new Name('John', 'Smith'),
                new Address('20 Main St', 'New York', '11001'),
                date '1960-8-22');
```

# Type Inheritance

- Suppose that we have the following type definition for people:

```
create type Person
        (name varchar(20),
         address varchar(20));
```

- We may want to store extra information in the database about people who are students, and about peoplewho are teachers.
- Since students and teachers are also people, we can use inheritance 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));
```

SAINTGITS
LEARN.GROW.EXCEL

# Type Inheritance

- 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 super type of Student, as well as of Teacher.
- The keyword final says that subtypes may not be created from the given type, while not final says that subtypes may be created.

# Table Inheritance

- Subtables in SQL correspond to the E-R notion of specialization/generalization.
- For instance, suppose we define the people table as follows.

```
create table people of Person;
```

- We can then define tables students and teachers as subtables of people, as follows.

```
create table students of Student
    under people;

create table teachers of Teacher
    under people;
```

- The typesof the subtables (Student and Teacher in the above example) are subtypes of the type of the parent table (Person in the above example).
- As a result, every attribute present in the table people is also present in the subtables students and teachers.
- Further, when we declare students and teachers as subtables of people, every tuple present in students or teachers becomes implicitly present in people.

# Table Inheritance

- Thus,if a query uses the table people, it will find not only tuples directly inserted into that table, but also tuples inserted into its subtables, namely students and teachers.
- However, only those attributes that are present in people can be accessed by that query.

**delete from** *people* **where** *P;*

- The above statement would delete all tuples from the table people, as well as its subtables students and teachers, that satisfy P.
- If the only keyword is added to the above statement, tuples that were inserted in subtables are not affected, even if they satisfy the where clause conditions.
- There are some consistency requirements for subtables.
- Before we state the constraints, we need a definition: we say that tuples in a subtable and parent table correspond if they have the same values for all inherited attributes.
- Thus, corresponding tuples represent the same entity.

SAINTGITS
LEARN.GROW.EXCEL

# Table Inheritance

- The consistency requirements for subtables are:

1. Each tuple of the supertable can correspond to at most one tuple in each of its immediate subtables.
2. SQL has an additional constraint that all the tuples corresponding to each other must be derived from one tuple (inserted into one table).

# Array and Multiset Types in SQL

- SQL supports two collection types: arrays and multisets.
- A multiset is an unordered collection, where an element may occur multiple times.
- Multisets are like sets, except that a set allows each element to occur at most once.
- The following example illustrates how these array and multiset-valued attributes can be defined in SQL.

```
create type Publisher as
        (name varchar(20),
         branch varchar(20));

create type Book as
        (title varchar(20),
         author_array varchar(20) array [10],
         pub_date date,
         publisher Publisher,
         keyword_set varchar(20) multiset);

create table books of Book;
```

SAINTGITS
LEARN.GROW.EXCEL

# Array and Multiset Types in SQL

- The first statement defines a type called Publisher with two components: a name and a branch.
- The second statement defines a structured type Book that contains a title, an author array, which is an array of up to 10 author names, a publication
- date, a publisher (of type Publisher), and a multiset of keywords.
- Finally, a table books containing tuples of type Book is created.

# Creating and Accessing Collection Values

- An array of values can be created in SQL:1999 in this way:

  **array['Silberschatz', 'Korth', 'Sudarshan']**

- Similarly, a multiset of keywords can be constructed as follows:

  **multiset['computer', 'database', 'SQL']**

- Thus,we can create a tuple of the type defined by the *books relation as:*

  ('Compilers', **array['Smith', 'Jones'], new *Publisher('McGraw-Hill', 'New York'),***

  **multiset['parsing', 'analysis'])**

- Here we have created a value for the attribute Publisher by invoking a constructor function for Publisher with appropriate arguments.
- Note that this constructor for Publisher must be created explicitly.
- If we want to insert the preceding tuple into the relation books, we could execute the statement:

```
insert into books
values ('Compilers', array['Smith', 'Jones'],
                     new Publisher('McGraw-Hill', 'New York'),
                     multiset['parsing', 'analysis']);
```

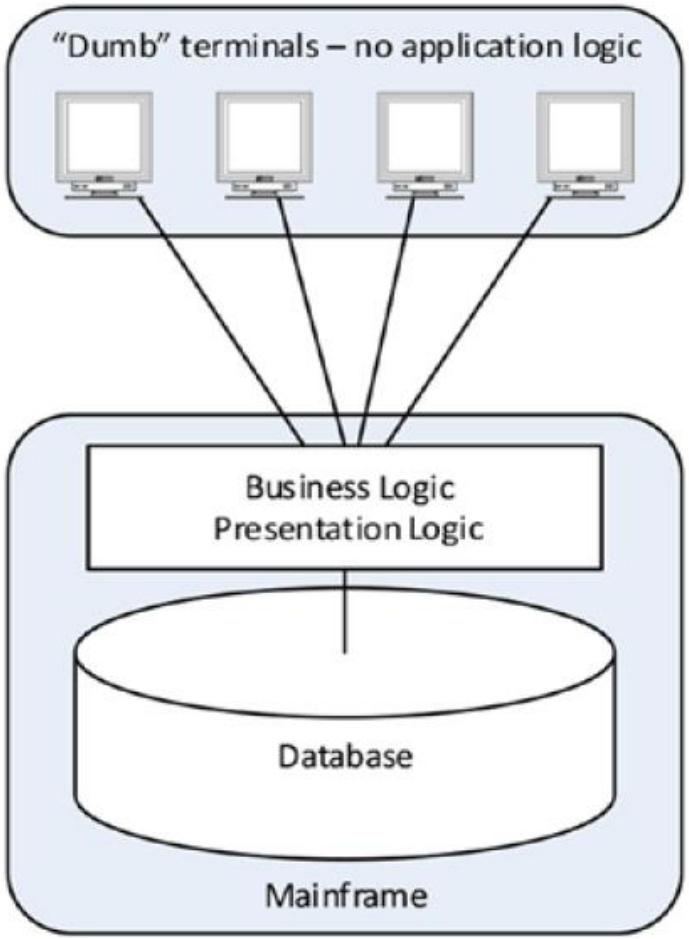
SAINTGITS
LEARN.GROW.EXCEL

# Distributed Relational Databases

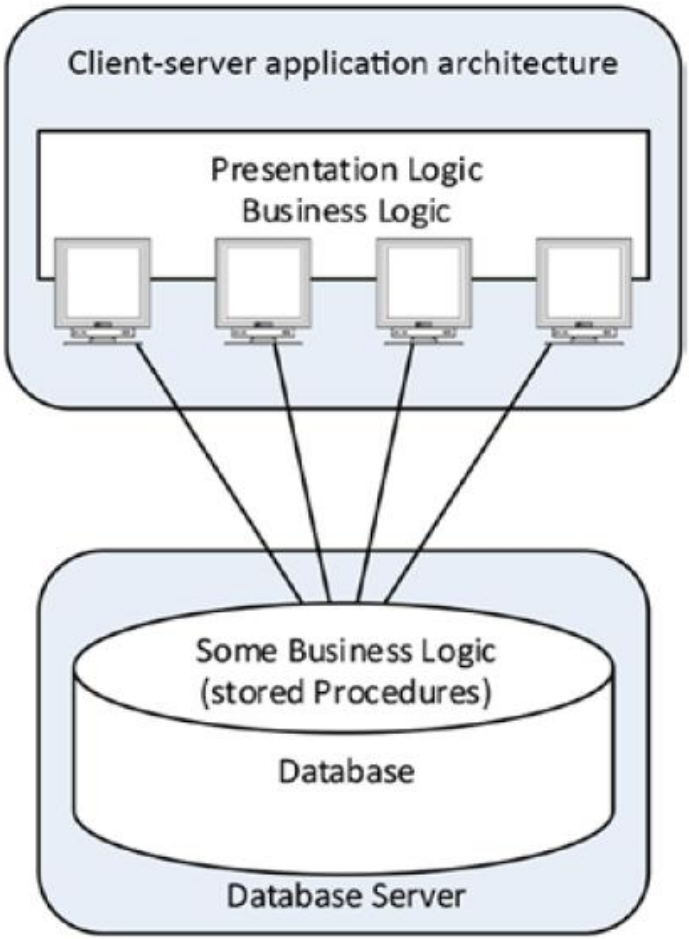
- The first database systems were designed to run on a single computer. Indeed, prior to the client-server revolution, all components of early database applications—including all the application code—would reside on a single system: the mainframe.
- In this centralized model, all program code runs on the server, and users communicate with the application code through dumb terminals (the terminals are “dumb” because they contain no application code).
- In the client-server model, presentation and business logic were implemented on workstations—usually Windows PCs—that communicated with a single back-end database server.
- In early Internet applications, business logic was implemented on one or more web application servers, while presentation logic was shared between the web browser and the application server, which still almost always communicated with a single database server.

SAINTGITS
LEARN.GROW.EXCEL

# Distributed Relational Databases



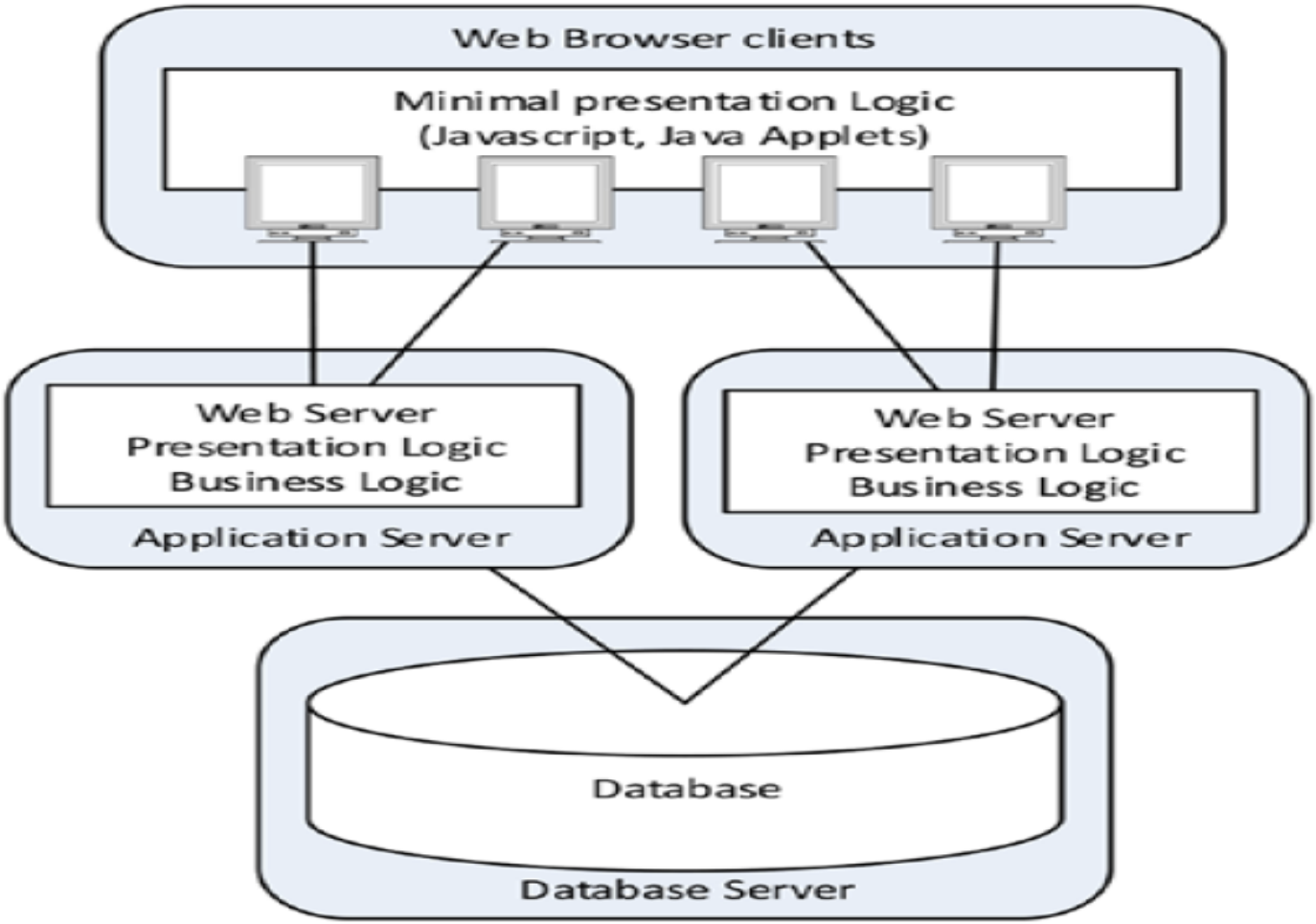

Mainframe application architecture




Client Server application architecture

SAINTGITS
LEARN.GROW.EXCEL

# Distributed Relational Databases




Web 1.0 application architecture

# Nonrelational Distributed Databases

- Maintaining ACID transactional integrity across multiple nodes in a distributed relational database is a significant challenge.
- However, in nonrelational database systems, ACID compliance is often not provided.
- For nonrelational distributed databases, the following considerations become more significant.
- **Balancing availability and consistency**: A distributed database that aims to scale beyond a single local network must choose between availability and consistency in the event of a network partition. An ACID-compliant database is obliged to favor consistency over all other factors. However, a nonrelational database without the constraint of strict ACID compliance can strike a different balance.
- **Hardware economics**: Even small differences in the cost of individual servers multiply quickly when a system scales to thousands or hundreds of thousands of nodes. Therefore, an economical database architecture will better leverage commodity hardware so as to take advantage of the best price/performance ratios available.
- **Resilience**:In a massive database cluster, nodes will fail from time to time. In the event of these failures, there can be no data loss, interruption to availability, or may be even failure at the transaction level.

# MongoDB Sharding and Replication

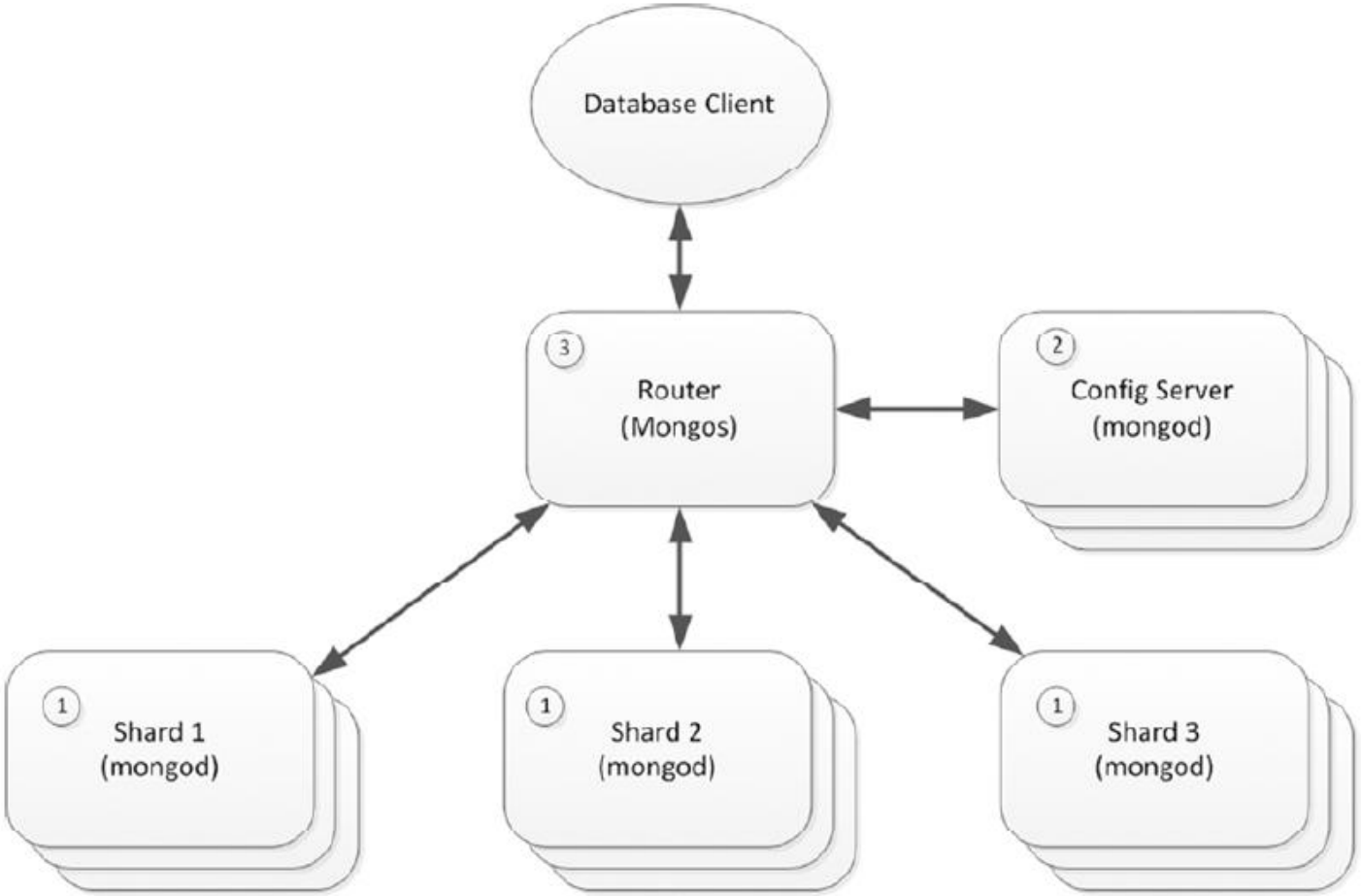
- MongoDB supports sharding to provide scale-out capabilities and replication for high availability.

**Sharding**

- A high-level representation of the MongoDB sharding architecture is shown in following figure.
- 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.

SAINTGITS
LEARN.GROW.EXCEL

# MongoDB Sharding and Replication




*MongoDB sharding architecture*

SAINTGITS
LEARN.GROW.EXCEL

# 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.
- 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 (unfilled orders or recent posts, for instance) evenly across the cluster, thus balancing the load more effectively.

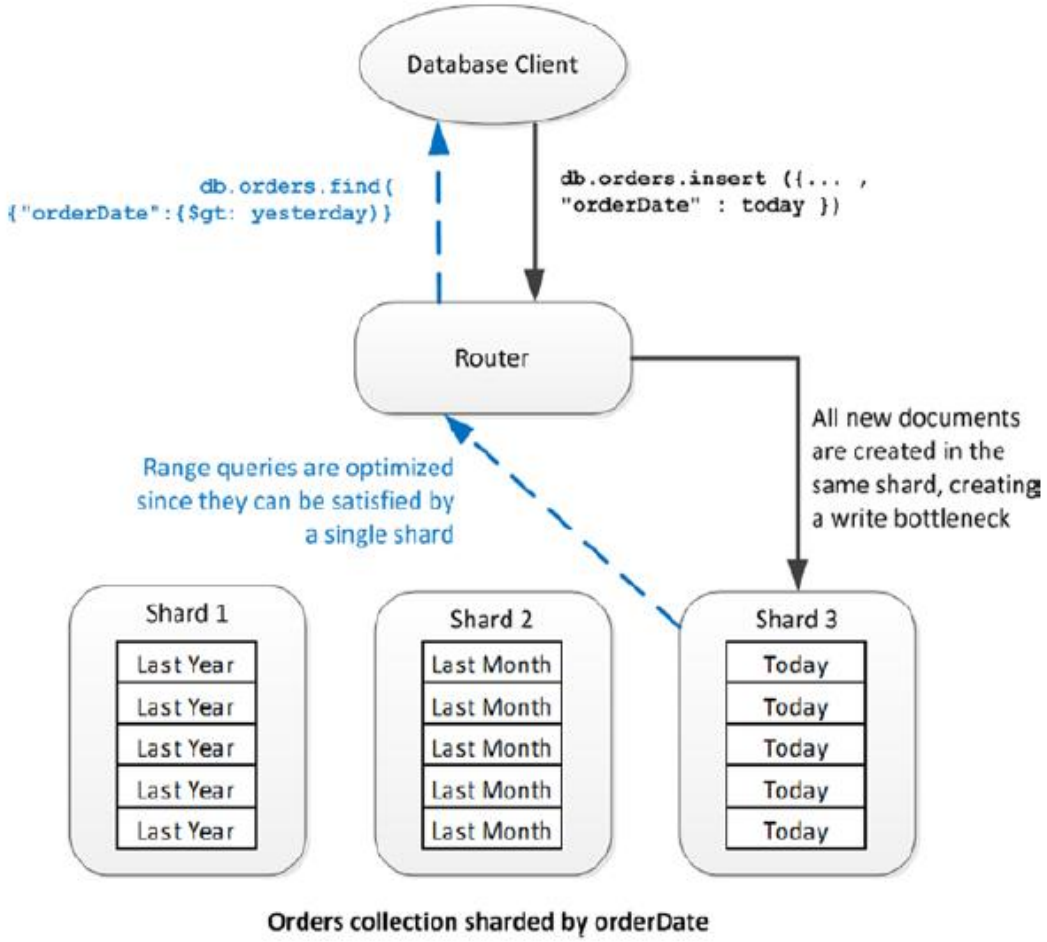
SAINTGITS
LEARN.GROW.EXCEL

# Sharding Mechanisms

- 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.

SAINTGITS
LEARN.GROW.EXCEL

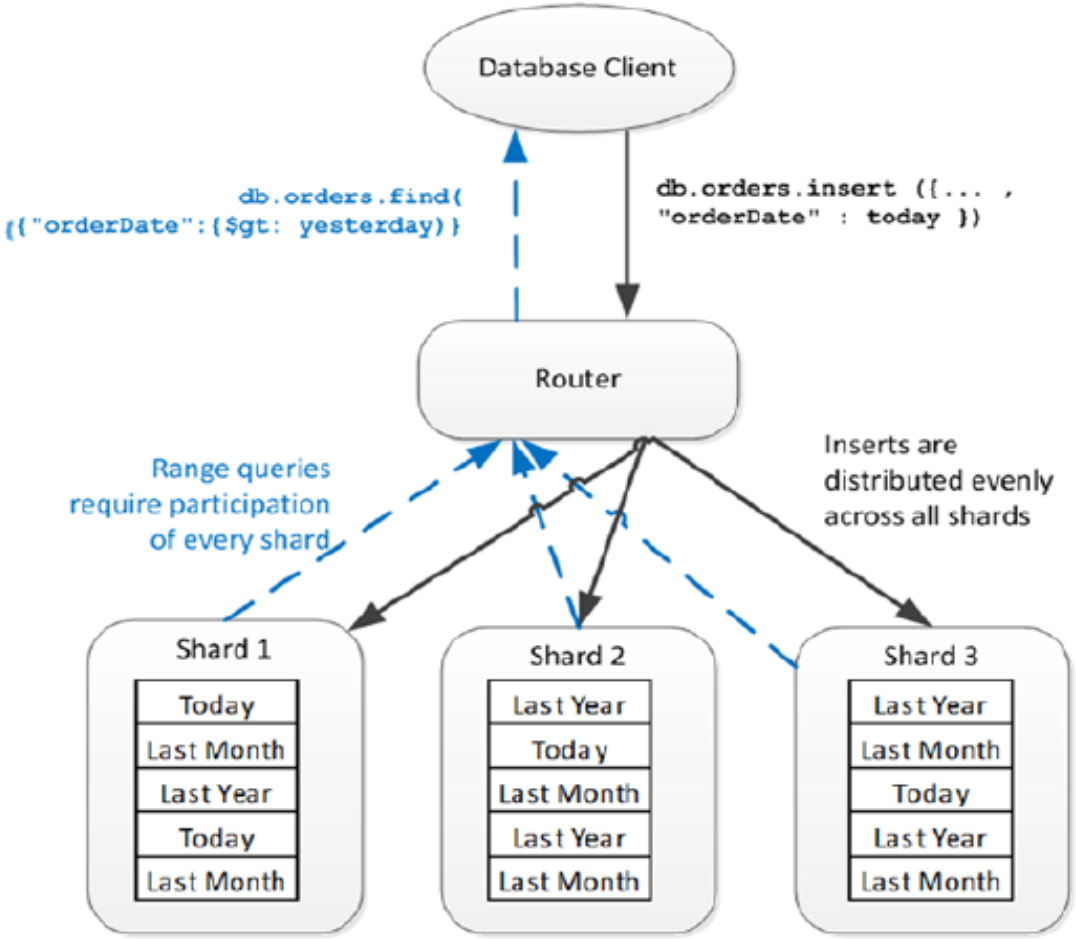
# Sharding Mechanisms




Orders collection sharded by orderDate

SAINTGITS
LEARN.GROW.EXCEL

# Sharding Mechanisms




**Orders collection sharded by Hashed orderDate**

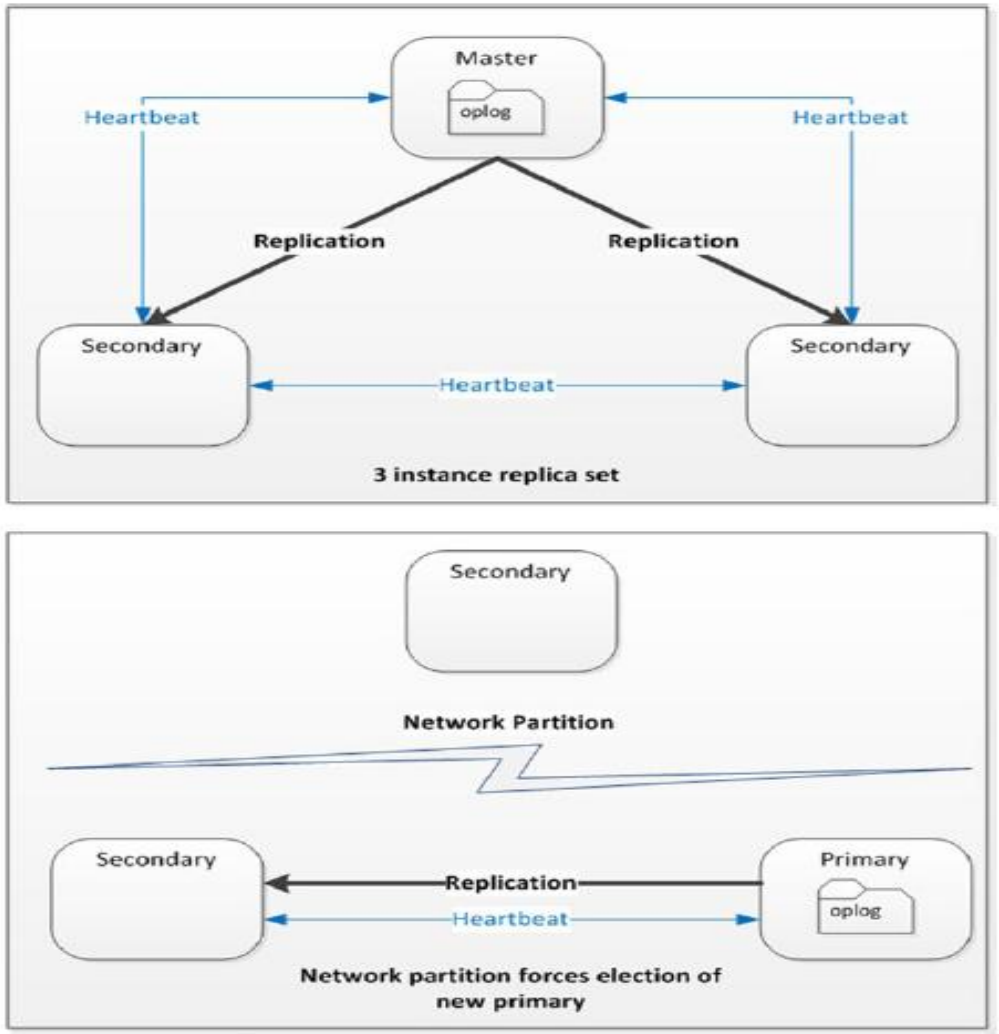
SAINTGITS
LEARN.GROW.EXCEL

# Replication

- Sharding is almost always combined with replication so as to ensure both availability and scalability in a production MongoDB deployment.
- 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 primary stores information about document changes in a collection within its local database, called the oplog.
- The primary will continuously attempt to apply these changes to secondary instances.
- 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.

SAINTGITS
LEARN.GROW.EXCEL

# Replication




*MongoDB replica set and primary failover*

SAINTGITS
LEARN.GROW.EXCEL

# Write Concern and Read Preference

- A MongoDB application has some control over the behavior of read and write operations, providing a degree of tunable consistency and availability.
- The **write concern** setting determines when MongoDB regards a write operation as having completed.
- By default, write operations complete once the primary has received the modification.
- This means that if the primary should fail irrecoverably, then data might be lost.
- To ensure that write operations have been propagated beyond the primary, the client can issue a blocking call, which will wait until the write has been received by all secondaries, a majority of secondaries, or a specified number of secondaries.
- The **read preference** determines where the client sends read requests.
- By default, all read requests are sent to the primary.
- However, the client driver can request that read requests be routed to the secondary if the primary is unavailable, or to secondaries, or to whichever server is “nearest.”
- The latter setting is intended to favor low latency over consistency.

SAINTGITS
LEARN.GROW.EXCEL

# MongoDB CRUD Operations

- CRUD operations create, read, update, and delete documents.

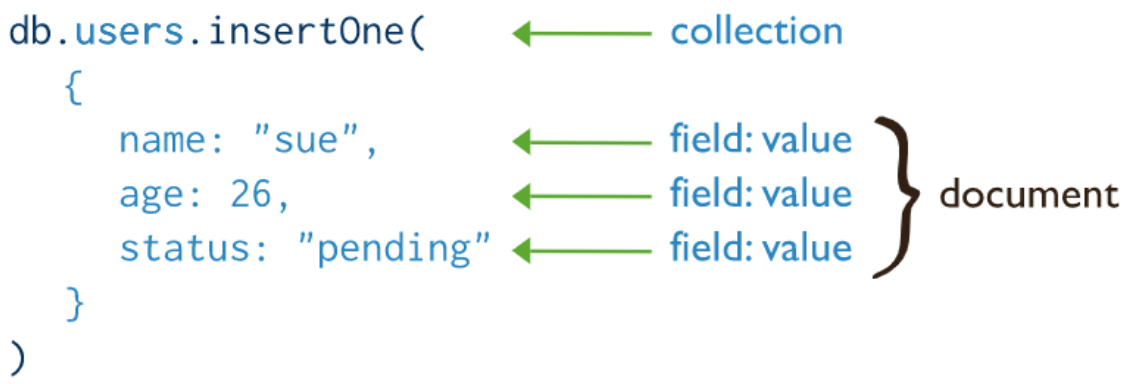
## Create Operations

Create or insert operations add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.

MongoDB provides the following methods to insert documents into a collection:

- `db.collection.insertOne()` *New in version 3.2*
- `db.collection.insertMany()` *New in version 3.2*

In MongoDB, insert operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.



SAINTGITS
LEARN.GROW.EXCEL

# MongoDB CRUD Operations

## Read Operations ¶

Read operations retrieve documents from a collection; i.e. query a collection for documents. MongoDB provides the following methods to read documents from a collection:

- `db.collection.find()`

You can specify query filters or criteria that identify the documents to return.

```
db.users.find(                          <-- collection
   { age: { $gt: 18 } },                <-- query criteria
   { name: 1, address: 1 }              <-- projection
).limit(5)                              <-- cursor modifier
```

SAINTGITS
LEARN.GROW.EXCEL

# MongoDB CRUD Operations

## Update Operations ¶

Update operations modify existing documents in a collection. MongoDB provides the following methods to update documents of a collection:

- `db.collection.updateOne()` *New in version 3.2*
- `db.collection.updateMany()` *New in version 3.2*
- `db.collection.replaceOne()` *New in version 3.2*

In MongoDB, update operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

You can specify criteria, or filters, that identify the documents to update. These filters use the same syntax as read operations.

```
db.users.updateMany(                    <-- collection
   { age: { $lt: 18 } },                <-- update filter
   { $set: { status: "reject" } }       <-- update action
)
```

SAINTGITS
LEARN.GROW.EXCEL

# MongoDB CRUD Operations

## Delete Operations ¶

Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection:

- `db.collection.deleteOne()` *New in version 3.2*
- `db.collection.deleteMany()` *New in version 3.2*

In MongoDB, delete operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

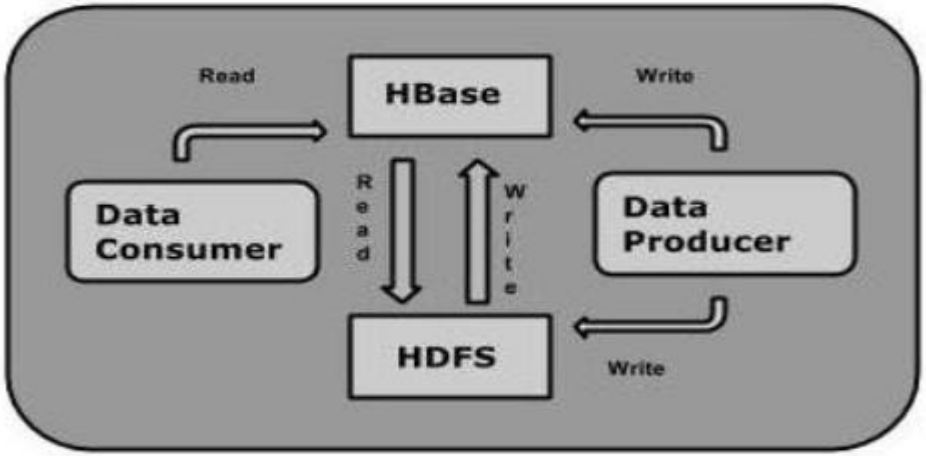
You can specify criteria, or filters, that identify the documents to remove. These filters use the same syntax as read operations.

```
db.users.deleteMany(          <-- collection
   { status: "reject" }       <-- delete filter
)
```

SAINTGITS
LEARN.GROW.EXCEL

# HBase

- HBase is a distributed column-oriented database built on top of the Hadoop file system.
- It is an open-source project and is horizontally scalable.
- HBase is a data model that is similar to Google's big table designed to provide quick random access to huge amounts of structured data.
- It leverages the fault tolerance provided by the Hadoop File System (HDFS).
- One can store the data in HDFS either directly or through HBase.
- Data consumer reads/accesses the data in HDFS randomly using HBase.
- HBase sits on top of the Hadoop File System and provides read and write access.



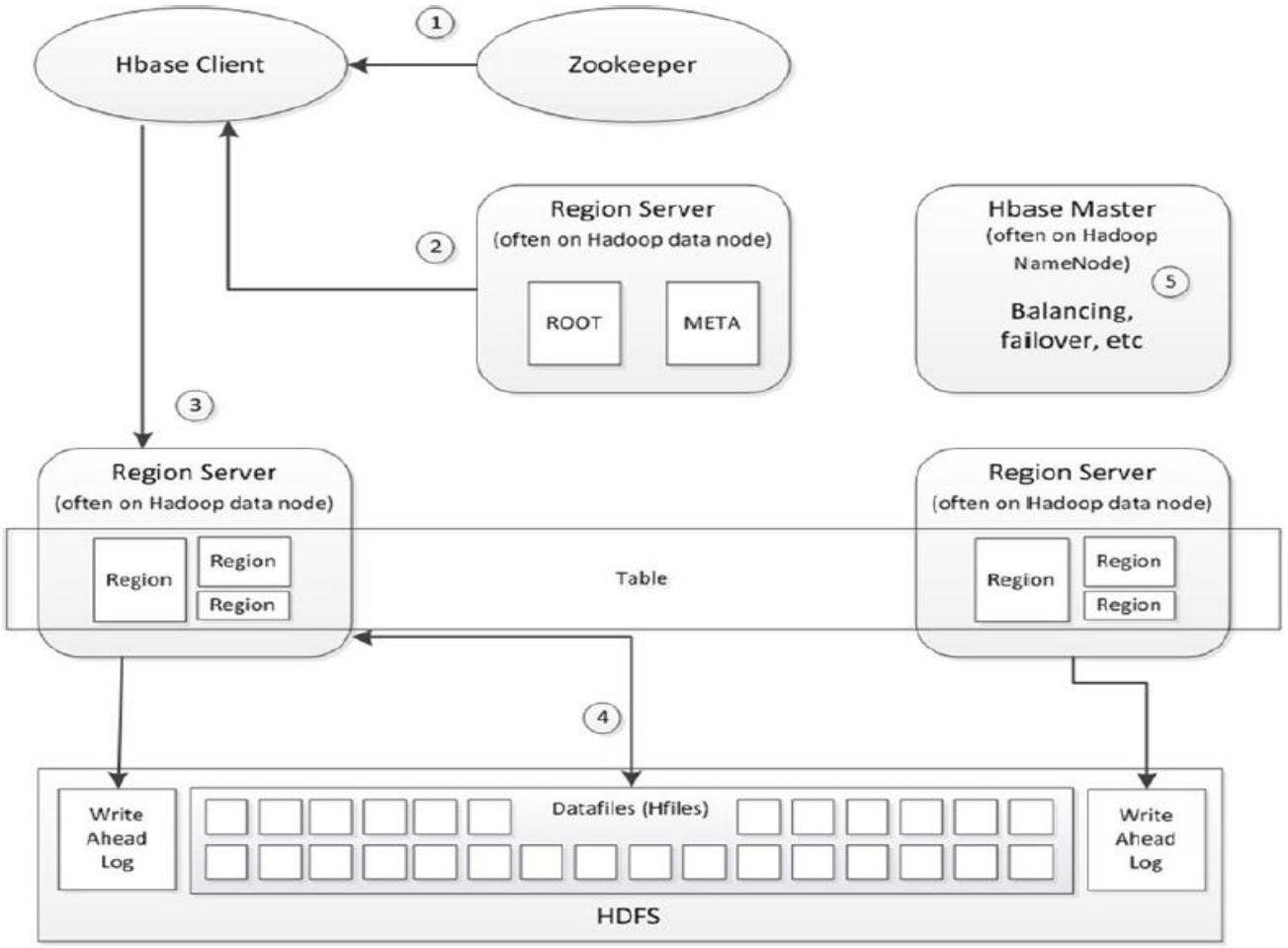
# Hbase - Tables, Regions, and RegionServers

- We can consider HBase tables as potentially massive tabular datasets that are implemented on disk by a variable number of HDFS files called Hfiles.
- 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.
- 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.

# Hbase - Tables, Regions, and RegionServers

- When an HBase client wishes to read or write to a specific key value, it will ask Zookeeper for the address of the RegionServer that controls the HBase catalog.
- This catalog consists of the tables -ROOT- and .META., which identify the RegionServers that are responsible for specific key ranges.
- The client will then establish a connection with that RegionServer and request to read or write the key value concerned.
- 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.
- Following figure 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).

# Hbase - Architecture




*HBase architecture*

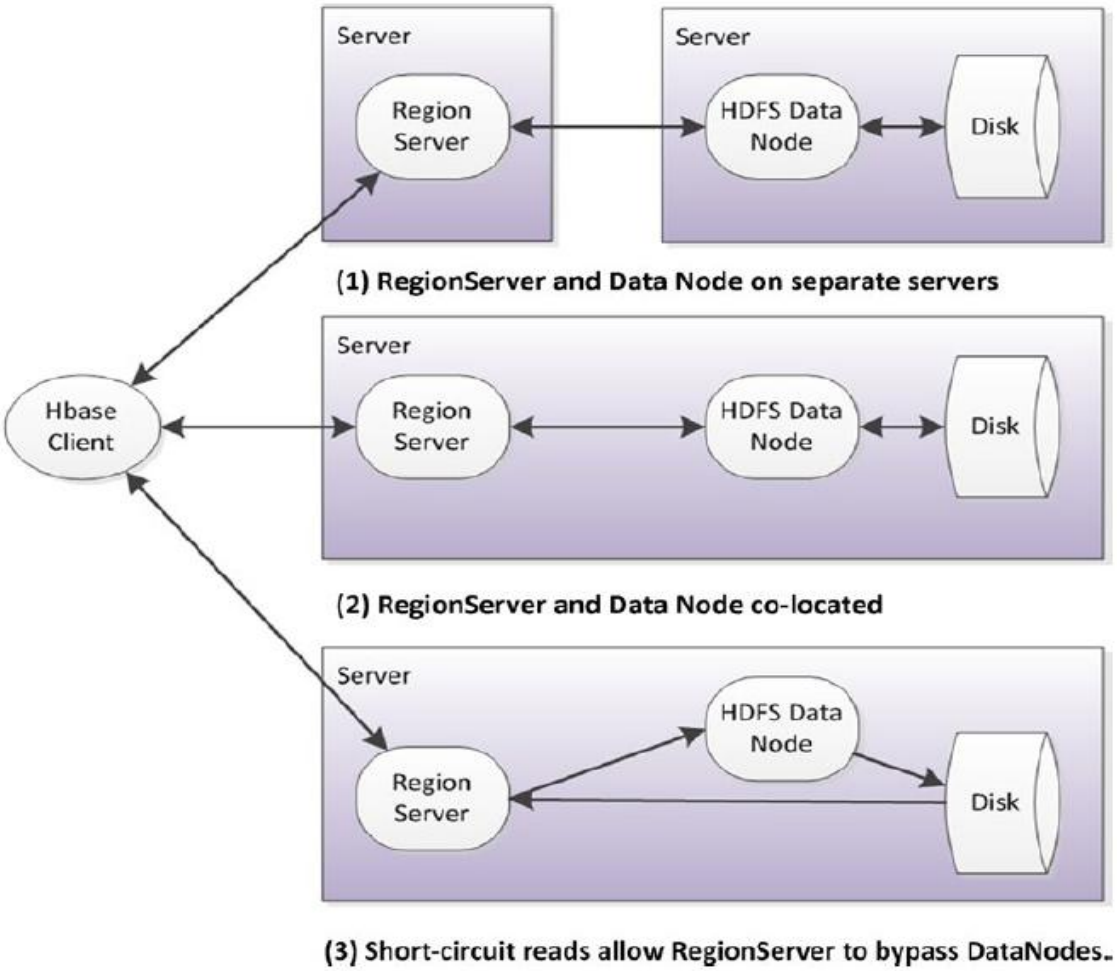
SAINTGITS
LEARN.GROW.EXCEL

# Hbase - Caching and Data Locality

- The RegionServer includes a block cache that can satisfy many reads from memory, and a 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.
- The RegionServer can act as a generic HDFS client, communicating with the HDFS NameNode to perform read and write operations to files.
- In the most typical production deployment scenario, 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 following figure.
- 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.

SAINTGITS
LEARN.GROW.EXCEL

# Hbase - Caching and Data Locality




*Data locality in HBase*

SAINTGITS
LEARN.GROW.EXCEL

# Cassandra

- Cassandra is a distributed database from Apache that is highly scalable and designed to manage very large amounts of structured data.
- It provides high availability with no single point of failure.
- Apache Cassandra is a highly scalable, high-performance distributed database designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure.
- It is a type of NoSQL database.
- A NoSQL database (sometimes called as Not Only SQL) is a database that provides a mechanism to store and retrieve data other than the tabular relations used in relational databases.
- These databases are schema-free, support easy replication, have simple API, eventually consistent, and can handle huge amounts of data.
- The primary objective of a NoSQL database is to have
  - simplicity of design,
  - horizontal scaling, and
  - finer control over availability.

SAINTGITS
LEARN.GROW.EXCEL

# NoSQL vs. Relational Database

| Relational Database | NoSql Database |
| --- | --- |
| Supports powerful query language. | Supports very simple query language. |
| It has a fixed schema. | No fixed schema. |
| Follows ACID (Atomicity, Consistency, Isolation, and Durability). | It is only “eventually consistent”. |
| Supports transactions. | Does not support transactions. |

# Cassandra

- Apache Cassandra is an open source, distributed and decentralized/distributed storage system (database), for managing very large amounts of structured data spread out across the world.
- It provides highly available service with no single point of failure.
- Listed below are some of the notable points of Apache Cassandra
  - It is scalable, fault-tolerant, and consistent.
  - It is a column-oriented database.
  - Its distribution design is based on Amazon's Dynamo and its data model on Google's Bigtable.
  - Created at Facebook, it differs sharply from relational database management systems.
  - Cassandra implements a Dynamo-style replication model with no single point of failure, but adds a more powerful "column family" data model.
  - Cassandra is being used by some of the biggest companies such as Facebook, Twitter, Cisco, Rackspace, ebay, Twitter, Netflix, and more.

SAINTGITS
LEARN.GROW.EXCEL

# Cassandra

- The design goal of Cassandra is to handle big data workloads across multiple nodes without any single point of failure.
- Cassandra has peer-to-peer distributed system across its nodes, and data is distributed among all the nodes in a cluster.
- All the nodes in a cluster play the same role.
- Each node is independent and at the same time interconnected to other nodes.
- Each node in a cluster can accept read and write requests, regardless of where the data is actually located in the cluster.
- When a node goes down, read/write requests can be served from other nodes in the network.

SAINTGITS
LEARN.GROW.EXCEL

# Cassandra

- Cassandra has become so popular because of its outstanding technical features. Given below are some of the features of Cassandra:
- **Elastic scalability** − Cassandra is highly scalable; it allows to add more hardware to accommodate more customers and more data as per requirement.
- **Always on architecture** − Cassandra has no single point of failure and it is continuously available for business-critical applications that cannot afford a failure.
- **Fast linear-scale performance** − Cassandra is linearly scalable, i.e., it increases your throughput as you increase the number of nodes in the cluster. Therefore it maintains a quick response time.
- **Flexible data storage** − Cassandra accommodates all possible data formats including: structured, semi-structured, and unstructured. It can dynamically accommodate changes to your data structures according to your need.
- **Easy data distribution** − Cassandra provides the flexibility to distribute data where you need by replicating data across multiple data centers.
- **Transaction support** − Cassandra supports properties like Atomicity, Consistency, Isolation, and Durability (ACID).
- **Fast writes** − Cassandra was designed to run on cheap commodity hardware. It performs blazingly fast writes and can store hundreds of terabytes of data, without sacrificing the read efficiency.

# Components of Cassandra

- The key components of Cassandra are as follows −
- **Node** − It is the place where data is stored.
- **Data center** − It is a collection of related nodes.
- **Cluster** − A cluster is a component that contains one or more data centers.
- **Commit log** − The commit log is a crash-recovery mechanism in Cassandra. Every write operation is written to the commit log.
- **Mem-table** − A mem-table is a memory-resident data structure. After commit log, the data will be written to the mem-table. Sometimes, for a single-column family, there will be multiple mem-tables.
- **SSTable** − It is a disk file to which the data is flushed from the mem-table when its contents reach a threshold value.
- **Bloom filter** − These are nothing but quick, nondeterministic, algorithms for testing whether an element is a member of a set. It is a special kind of cache. Bloom filters are accessed after every query.

SAINTGITS
LEARN.GROW.EXCEL

# Cassandra – Gossip Protocol

- In HBase and MongoDB, we encountered the concept of master nodes—nodes which have a specialized supervisory function, coordinate activities of other nodes, and record the current state of the database cluster.
- In Cassandra and other Dynamo databases, 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.
- 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 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 is aptly named: when people gossip, they generally tend to gossip about other people! Likewise, in Cassandra, the nodes gossip about other nodes as well as about their own state.

# Cassandra – Consistent Hashing

- Cassandra and other dynamo-based 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:

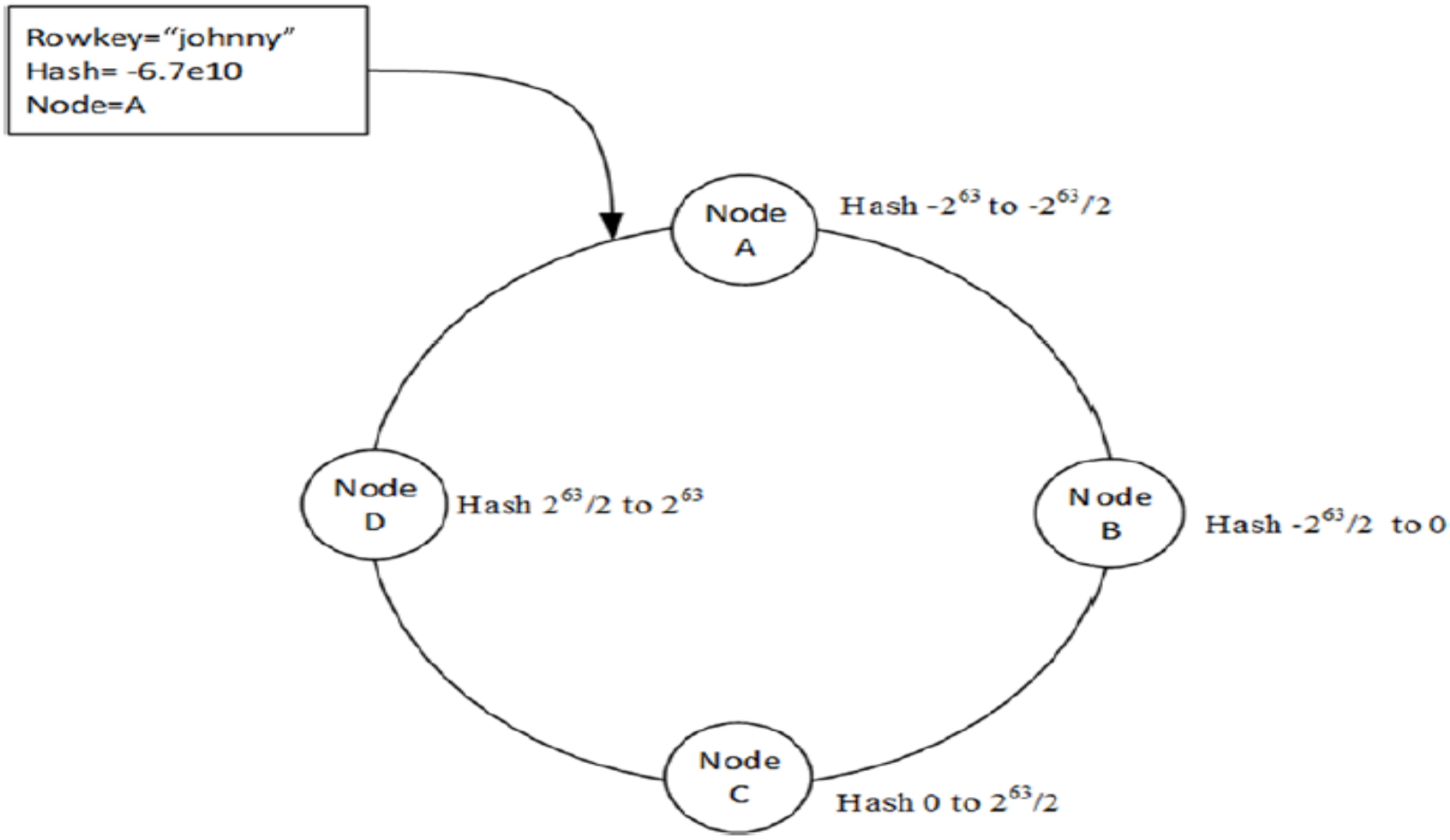
| Node | Low Hash | High Hash |
|---|---|---|
| Node A | $-2^{63}$ | $-2^{63}/2$ |
| Node B | $-2^{63}/2$ | 0 |
| Node C | 0 | $2^{63}/2$ |
| Node D | $2^{63}/2$ | $2^{63}$ |

# Cassandra – Consistent Hashing

- We usually visualize the cluster as a ring: the circumference of the ring represents all the possible hash values, and the location of the node on the ring represents its area of responsibility.
- Following figure 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.

SAINTGITS
LEARN.GROW.EXCEL

# Cassandra – Consistent Hashing

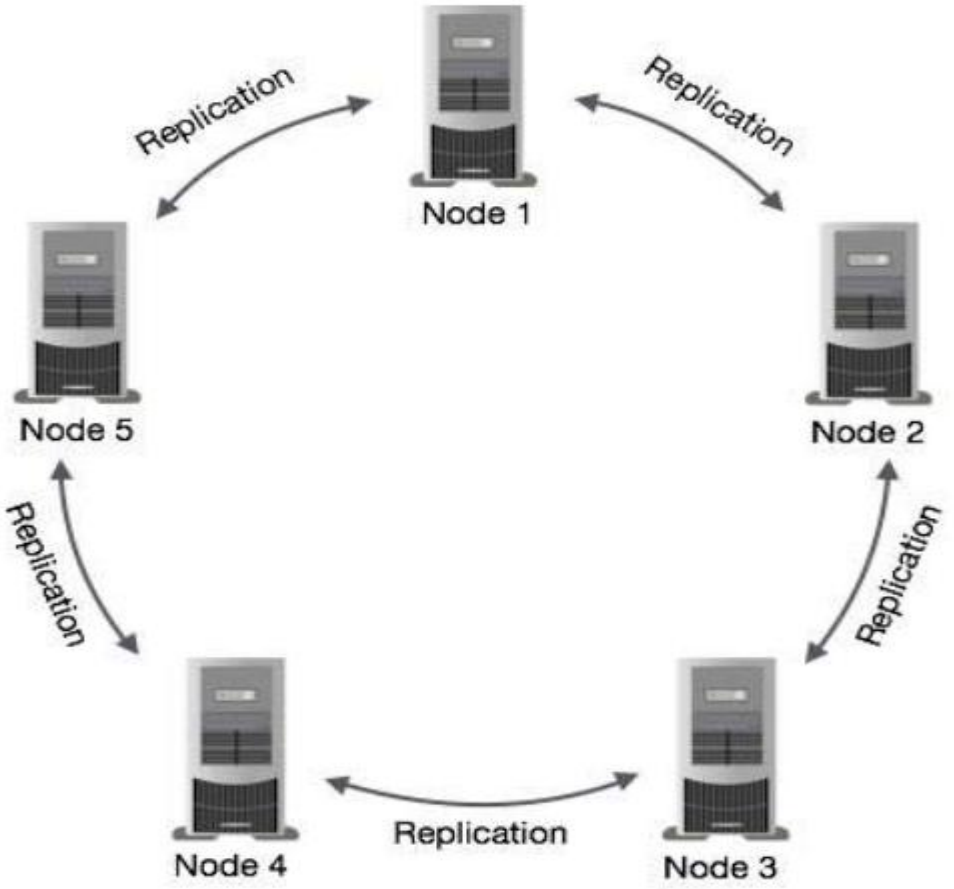



*Consistent hashing*

# Cassandra – Data Replication

- In Cassandra, one or more of the nodes in a cluster act as replicas for a given piece of data.
- If it is detected that some of the nodes responded with an out-of-date value, Cassandra will return the most recent value to the client.
- After returning the most recent value, Cassandra performs a **read repair** in the background to update the stale values.
- The following figure shows a schematic view of how Cassandra uses data replication among the nodes in a cluster to ensure no single point of failure.

# **Cassandra –** Data Replication




- Cassandra uses the **Gossip Protocol** in the background to allow the nodes to communicate with each other and detect any faulty nodes in the cluster.

SAINTGITS
LEARN.GROW.EXCEL

# CAP Theorm

- The CAP theorem says that in a distributed database system, you 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. Whenever you read a record (or data), consistency guaranties that it will give same data how many times you read. Simply we can say that each server returns the right response to each request, thus the system will be always consistent whenever you read or write data into that.
- **Availability** means that in the event of a failure, the database remains operational. Availability simply means that each request eventually receives a response (even if it's not the latest data or consistent across the system or just a message saying the system isn't working).
- **Partition tolerance** means that the database can maintain operations in the event of the network's failing between two segments of the distributed system. means that the cluster continues to function even if there is a "partition" (communications break) between two nodes (both nodes are up, but can't communicate).
- One property should be scarified among three, so you have to choose combination of **CA or CP or AP.**

SAINTGITS
LEARN.GROW.EXCEL

# CAP Theorm