Module v

Distributed Databases

- A distributed database is basically a database that is not limited to one system, it is spread over different sites, i.e, on multiple computers or over a network of computers.
- A distributed database system is located on various sites that don't share physical components. This may be required when a particular database needs to be accessed by various users globally.
- It needs to be managed such that for the users it looks like one single database.

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 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.

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.

- **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.

- 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.

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.
- the account relation can be divided into several different fragments, each of which consists of tuples of accounts belonging to a particular branch.

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

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

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

- 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 is the same as decomposition. Vertical fragmentation of r(R) involves the definition of several subsets of attributes R1, R2, . . . , Rn of the schema R so that:
  $R = R1 \cup R2 \cup \cdots \cup R$
- Each fragment r_i of r is defined by

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

- we can reconstruct relation r from the fragments by taking the natural join:

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

- One way of ensuring that the relation r can be reconstructed is to include the primary-key attributes of R in each Ri . More generally, any superkey can be used.
  It is often convenient to add a special attribute, called a tuple-id, to the schema R

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, 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.

- Data items—such as relations, fragments, and replicas—must have unique names. This property is easy to ensure in a centralized database. In a distributed database, however, we must take care to ensure that two sites do not use the same name for distinct data items.
- One solution to this problem is to require all names to be registered in a central **name server**. The name server helps to ensure that the same name does not get used for different data items.

- The database system can create a set of alternative
names, or aliases, for data items. A user may thus refer to data items by simple names that are translated by the system to complete names.

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.

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.
- each site contains two subsystems
- The **transaction manager** manages the execution of those transactions (or subtransactions) that access data stored in a local site.
- The transaction coordinator coordinates the execution of the various transactions (both local and global) initiated at that site.

- 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.

- the coordinator is responsible for:
  - Starting the execution of the transaction.
  - Breaking the transaction into a number of sub transactions 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 Failure Modes

- A distributed system may suffer from the same types of failure that a centralized system does
- • Failure of a site.
  - Loss of messages.
  - Failure of a communication link.
  - Network partition.

Object-BasedDatabases

- complex application domains require correspondingly complex data types, such as nested record structures, multivalued attributes, and inheritance, which are supported by traditional programming languages.
- The object-relational data model extends the relational data model by providing a richer type system including complex data types and object orientation.

- Object-relational database systems, that is, database systems based on the object-relation model, provide a convenient migration path for users
of relational databases who wish to use object-oriented features.

- Two approaches are used
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.

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.
- 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.
- With complex type systems we can represent E-R model concepts, such as composite attributes, multivalued attributes, generalization, and specialization
directly, without a complex translation to the relational model.

- Rather than view a database as a set of records, users of certain applications view it as a set of objects (or entities).

Structured Types and Inheritance in SQL

- 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;
```

- 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;
```

- 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.
- 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 alternative way of defining composite attributes in SQL is to use unnamed row types.

```
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);
```

- This definition is equivalent to the preceding table definition, except that the attributes name and address have unnamed types, and the rows of the table also have an unnamed type

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

```
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
```

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

- we could declare a constructor for the type Name like this

```
ate function Name (firstname varchar(20), lastname varchar(20
urns Name
gin
   set self.firstname = firstname;
   set self.lastname = lastname;
d
```

- We can then use new Name('John', 'Smith') to create a value of the type Name.
  We can construct a row value by listing its attributes within parentheses.
- By default every structured type has a constructor with no arguments, which sets the attributes to their default values. Any other constructors have to be created explicitly. There can be more than one constructor for the same structured type; although they have the same name, they must be distinguishable by the number of arguments and types of their arguments

- 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 people who are teachers. Since students and teachers are alsopeople, 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));
```

- 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 .
- Multiple inheritance

```
create type TeachingAssistant
        under Student, 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.

Table Inheritance

- create table people of Person;
- We can then define tables students and teachers as subtables of people,

```
create table students of Student
    under people;

create table teachers of Teacher
    under people;
```

- Further, when we declare students and teachers as subtables of people, everytuple present in students or teachers becomes implicitly present in people. 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.

- SQL permits us to find tuples that are in people but not in its subtables by using “only people”in place of people in a query. The only keyword can also be used in delete and update statements. Without the only keyword, a delete statement on a supertable, such as people, also deletes tuples that were originally inserted in subtables (such as students);
- delete from people where P;
- 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
- multiple inheritance is possible with tables,

```
create table teaching_assistants
of TeachingAssistant
      under students, teachers;
```

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.

```
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;
```

Creating and Accessing Collection Values

- An array of values can be created in SQL:

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

- Similarly, a multiset of keywords can be constructed as

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

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

```
ompilers', array['Smith', 'Jones'], new Publisher('McGraw-Hill', 'New York'),
                                    multiset['parsing', 'analysis'])
```

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

Object-Identity and Reference Types in SQL

- Object-oriented languages provide the ability to refer to objects. An attribute of a type can be a reference to an object of a specified type.
- For example, in SQL we can define a type Department with a field name and a field head that is a reference to the type Person, and a table departments of type Department, as follows:

```
create type Department (
        name varchar(20),
        head ref(Person) scope people);

create table departments of Department;
```

- We can omit the declaration scope people from the type declaration and instead
make an addition to the create table statement

```
create table departments of Department
        (head with options scope people);
```

- The referenced table must have an attribute that stores the identifier of the tuple. We declare this attribute, called the self-referential attribute, by adding a ref is clause to the create table statement:

Next generation databases

- CAP theorem
- The CAP Theorem is comprised of three components (hence its name) as they relate to distributed data stores:
- **Consistency.** All reads receive the most recent write or an error.
- **Availability.** All reads contain data, but it might not be the most recent.
- **Partition tolerance.** The system continues to operate despite network failures (ie; dropped partitions, slow network connections, or unavailable network connections between nodes.)

- In normal operations, your data store provides all three functions. But the CAP theorem maintains that when a distributed database experiences a network failure, you 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.

- if Partition means a break in communication then Partition tolerance would mean that the system should still be able to work even if there is a partition in the system. Meaning if a node fails to communicate, then one of the replicas of the node should be able to retrieve the data required by the user.
- The CAP theorem states that a distributed database system has to make a tradeoff between Consistency and Availability when a Partition occurs.

- 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.

Non-relational database

- Non-relational databases (often called NoSQL databases) are different from traditional relational databases in that they store their data in a non-tabular form.
- Instead, non-relational databases might be based on data structures like documents. A document can be highly detailed while containing a range of different types of information in different formats.

- There are several advantages to using non-relational databases, including:
- **Massive dataset organization**
  In the age of Big Data, non-relational databases can not only store massive quantities of information, but they can also query these datasets with ease. Scale and speed are crucial advantages of non-relational databases.

- **Flexible database expansion**
  Data is not static. As more information is collected, a non-relational database can absorb these new data points, enriching the existing database with new levels of granular value even if they don't fit the data types of previously existing information.
- **Multiple data structures**
- **Built for the cloud**

MongoDB

- MongoDB is an open-source document database and leading NoSQL database.
- MongoDB works on concept of collection and document.
- Rather than using the tables and fixed schemas of a relational database management system (RDBMS), MongoDB uses key-value storage in the collection of documents. It also supports a number of options for horizontal scaling in large, production environments.

- MongoDB is a NoSQL document database system that scales well horizontally and implements data storage through a key-value system.

MongoDB Sharding

- MongoDB achieves scaling through a technique known as "sharding". It is the process of writing data across different servers to distribute the read and write load and data storage requirements
- Sharding is the process of storing data records across multiple machines and it is MongoDB's approach to meeting the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput

- Sharding solves the problem with horizontal scaling. With sharding, you add more machines to support data growth and the demands of read and write operations.

MongoDB Replication

- Replica Sets are a great way to replicate MongoDB data across multiple servers and have the database automatically failover in case of server failure.

MongoDB sharding basics

- MongoDB sharding works by creating a cluster of MongoDB instances consisting of at least three servers. That means sharded clusters consist of three main components:
- The shard
- Mongos
- Config servers

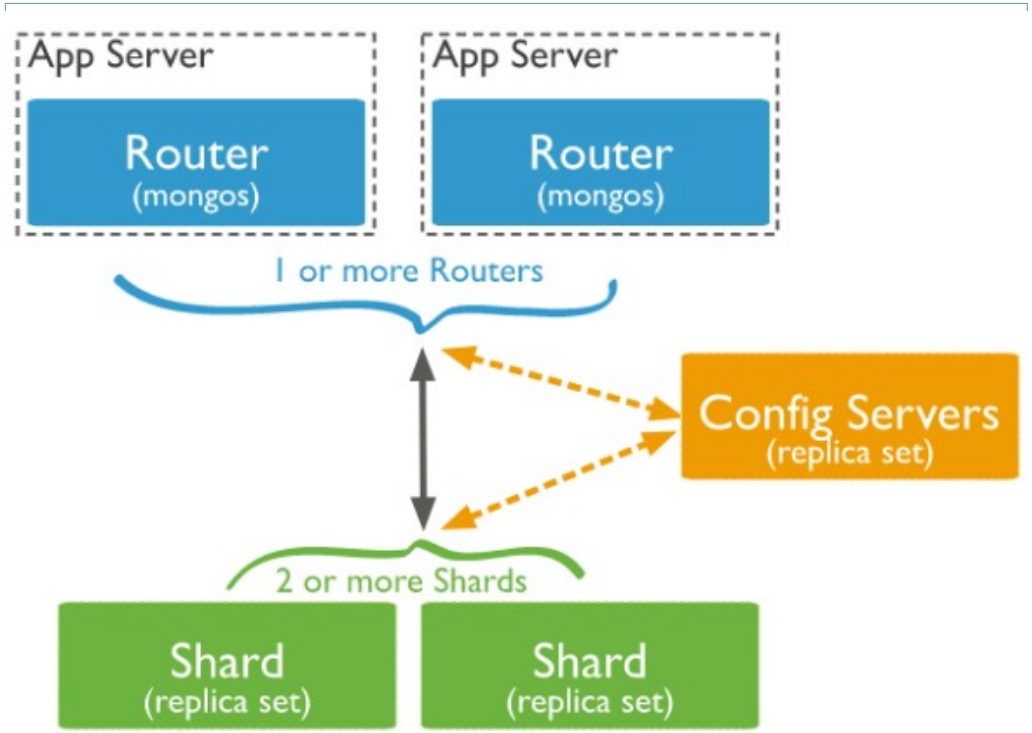
Shard

- A shard is a single MongoDB instance that holds a subset of the sharded data. Shards can be deployed as replica sets to increase availability and provide redundancy.

  The combination of multiple shards creates a complete data set. For example, a 2 TB data set can be broken down into four shards, each containing 500 GB of data from the original data set.

- **Mongos**
- Mongos act as the query router providing a stable interface between the application and the sharded cluster. This MongoDB instance is responsible for routing the client requests to the correct shard.

- **Config Servers**
- Configuration servers store the metadata and the configuration settings for the whole cluster.



- The application communicates with the routers (mongos) about the query to be executed.
- The mongos instance consults the config servers to check which shard contains the required data set to send the query to that shard.
- Finally, the result of the query will be returned to the application.

HBase

- HBase is a column-oriented non-relational database management system that runs on top of Hadoop Distributed File System (HDFS). HBase provides a fault-tolerant way of storing sparse data sets, which are common in many big data use cases. It is well suited for real-time data processing or random read/write access to large volumes of data.

- Unlike relational database systems, HBase does not support a structured query language like SQL; in fact, HBase isn't a relational data store at all. HBase applications are written in Java much like a typical Apache MapReduce application.

- HBase is a **column-oriented database** and the tables in it are sorted by row. The table schema defines only column families, which are the key value pairs. A table have multiple column families and each column family can have any number of columns. Subsequent column values are stored contiguously on the disk. Each cell value of the table has a timestamp

- in an HBase:
- Table is a collection of rows.
- Row is a collection of column families.
- Column family is a collection of columns.
- Column is a collection of key value pairs.

Base and RDBMS

HBase	RDBMS
HBase is schema-less, it doesn't have the concept of fixed columns schema; defines only column families.	An RDBMS is governed by its schema, which describes the whole structure of tables.
It is built for wide tables. HBase is horizontally scalable.	It is thin and built for small tables. Hard to scale.
No transactions are there in HBase.	RDBMS is transactional.
It has de-normalized data.	It will have normalized data.
It is good for semi-structured as well as structured data.	It is good for structured data.

Features of HBase

- HBase is linearly scalable.
- It has automatic failure support.
- It provides consistent read and writes.
- It integrates with Hadoop, both as a source and a destination.
- It has easy java API for client.
- It provides data replication across clusters

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.

- 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

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

- **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

Cassandra Query Language

- users can access Cassandra through its nodes using Cassandra Query Language (CQL). CQL treats the database **(Keyspace)** as a container of tables. Programmers use **cqlsh:** a prompt to work with CQL or separate application language drivers.

- **Write Operations**
- Every write activity of nodes is captured by the **commit logs** written in the nodes. Later the data will be captured and stored in the **mem-table.** Whenever the mem-table is full, data will be written into the **SStable** data file. All writes are automatically partitioned and replicated throughout the cluster. Cassandra periodically consolidates the SSTables, discarding unnecessary data.

- **Read Operations**
- During read operations, Cassandra gets values from the mem-table and checks the bloom filter to find the appropriate SSTable that holds the required data.

RDBMS	Cassandra
RDBMS deals with structured data.	Cassandra deals with unstructured data.
It has a fixed schema.	Cassandra has a flexible schema.
In RDBMS, a table is an array of arrays. (ROW x COLUMN)	In Cassandra, a table is a list of "nested key-value pairs". (ROW x COLUMN key x COLUMN value)
Database is the outermost container that contains data corresponding to an application.	Keyspace is the outermost container that contains data corresponding to an application.
Tables are the entities of a database.	Tables or column families are the entity of a keyspace.
Row is an individual record in RDBMS.	Row is a unit of replication in Cassandra.
Column represents the attributes of a relation.	Column is a unit of storage in Cassandra.
RDBMS supports the concepts of foreign keys, joins.	Relationships are represented using collections.