UNIT I DISTRIBUTED DATABASES

Distributed Systems – Introduction – Architecture – Distributed Database Concepts – Distributed Data Storage – Distributed Transactions – Commit Protocols – Concurrency Control – Distributed Query Processing

Distributed Systems and introduction

A **distributed computing system** consists of a number of processing sites or nodes that are interconnected by a computer network and that cooperate in performing certain assigned tasks. As a general goal, distributed computing systems partition a big, unmanageable problem into smaller pieces and solve it efficiently in a coordinated manner. Thus, more computing power is harnessed to solve a complex task, and the autonomous processing nodes can be managed independently while they cooperate to provide the needed functionalities to solve the problem. DDB technology resulted from a merger of two technologies: database technology and distributed systems technology.

Database is stored on several computers that communicate via media such as wide-area networks, telephone lines, or local area networks.
_ Appears to user as a single system
_ Processes complex queries
_ Processing may be done at a site other than the initiator of the request
_ Transaction management
_ Optimization of queries provided automatically

Several distributed database prototype systems were developed in the 1980s and 1990s to address the issues of data distribution, data replication, distributed query and transaction processing, distributed database metadata management, and other topics. More recently, many new technologies have emerged that combine distributed and database technologies. These technologies and systems are being developed for dealing with the storage, analysis, and mining of the vast amounts of data that are being produced and collected, and they are referred to generally as **big data technologies**. The origins of big data technologies come from distributed systems and database systems, as well as data mining and machine learning algorithms that can process these vast amounts of data to extract needed knowledge.

Distributed Database Architectures

The parallel architecture is more common in high-performance computing, where there is a need for multiprocessor architectures to cope with the volume of data undergoing transaction processing and warehousing applications. We then introduce a generic architecture of a distributed database. This is followed by discussions on the architecture of three-tier client/server and federated database systems.

Parallel versus Distributed Architectures

There are two main types of multiprocessor system architectures that are commonplace:

■ **Shared memory (tightly coupled) architecture.** Multiple processors share secondary (disk) storage and also share primary memory.

■ **Shared disk (loosely coupled) architecture.** Multiple processors share secondary (disk) storage but each has their own primary memory.

These architectures enable processors to communicate without the overhead of exchanging messages over a network.[4] Database management systems developed using the above types of architectures are termed **parallel database management systems** rather than DDBMSs, since they utilize parallel processor technology. Another type of multiprocessor architecture is called **shared-nothing architecture**. In this architecture, every processor has its own primary and secondary (disk) memory, no common memory exists, and the processors communicate over a highspeed interconnection network (bus or switch). Although the shared-nothing architecture resembles a distributed database computing environment, major differences exist in the mode of operation. In shared-nothing multiprocessor systems, there is symmetry and homogeneity of nodes; this is not true of the distributed database environment, where heterogeneity of hardware and operating system at each node is very common. Shared-nothing architecture is also considered as an environment for parallel databases. Figure 23.7(a) illustrates a parallel database (shared nothing), whereas Figure 23.7(b) illustrates a centralized database with distributed access and Figure 23.7(c) shows a pure distributed database. We will not expand on parallel architectures and related data management issues here.

General Architecture of Pure Distributed Databases

In this section, we discuss both the logical and component architectural models of a DDB. In Figure 23.8, which describes the generic schema architecture of a DDB, the enterprise is presented with a consistent, unified view showing the logical structure of underlying data across all nodes. This view is represented by the global conceptual schema (GCS), which provides network transparency (see Section 23.1.2). To accommodate potential heterogeneity in the DDB, each node is shown as having its own local internal schema (LIS) based on physical organization details at that particular site. The logical organization of data at each site is specified by the local conceptual schema (LCS). The GCS, LCS, and their underlying mappings provide the fragmentation and replication transparency discussed in Section 23.1.2. Figure 23.8 shows the component architecture of a DDB. It is an extension of its centralized counterpart (Figure 2.3) in Chapter 2. For the sake of simplicity, common elements are not shown here. The global query compiler references the global conceptual schema from the global system catalog to verify and impose defined constraints. The global query optimizer references both global and local conceptual schemas and generates optimized local queries from global queries. It evaluates all candidate strategies using a cost function that estimates cost based on response time (CPU, I/O, and network latencies) and estimated sizes of intermediate results. The latter is particularly important in queries involving joins. Having computed the cost for each candidate, the optimizer selects the candidate with the minimum cost for execution. Each local DBMS would have its local query optimizer, transaction manager, and execution engines as well as the local system catalog, which houses the local schemas. The global transaction manager is responsible for coordinating the execution across multiple sites in conjunction with the local transaction manager at those sites.

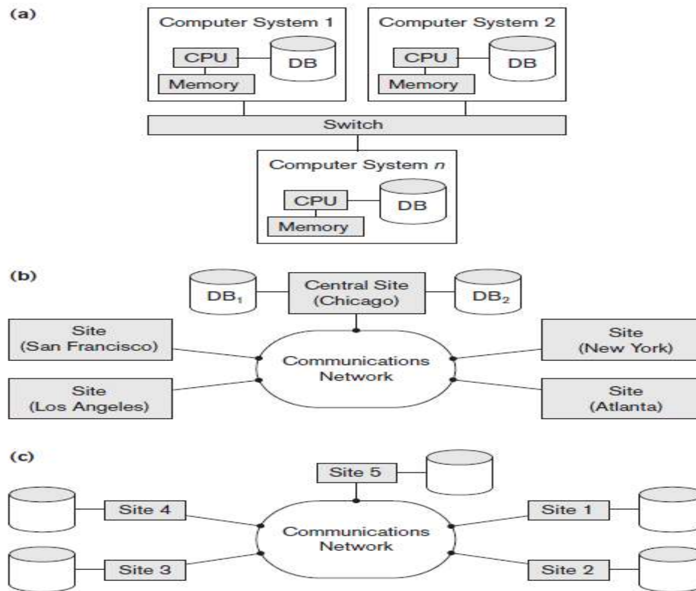


Figure 23.7
Some different database system architectures. (a) Shared-nothing architecture. (b) A networked architecture with a centralized database at one of the sites. (c) A truly distributed database architecture.

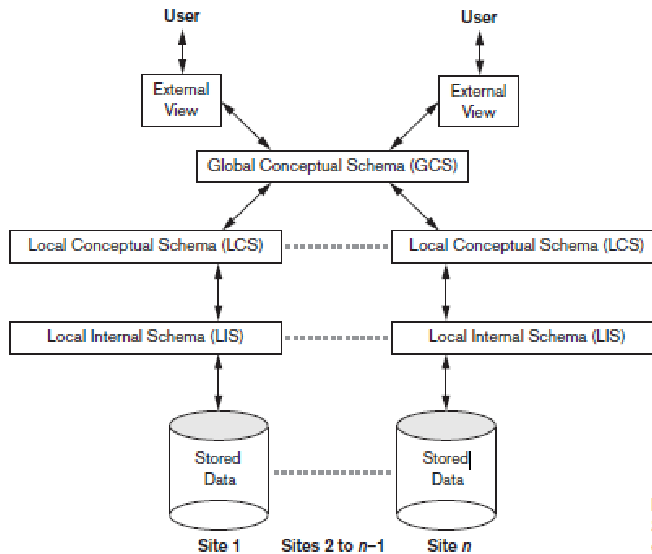


Figure 23.8
Schema architecture of distributed databases.

Federated Database Schema Architecture

Typical five-level schema architecture to support global applications in the FDBS environment is shown in Figure 23.9. In this architecture, the **local schema** is the conceptual schema (full database definition) of a component database, and the **component schema** is derived by translating the local schema into a canonical data model or common data model (CDM) for the FDBS. Schema translation from the local schema to the component schema is accompanied by generating mappings to transform commands on a component schema into commands on the corresponding local schema. The **export schema** represents the subset of a component schema that is available to the FDBS. The **federated schema** is the global schema or view, which is the result of integrating all the shareable export schemas. The **external schemas** define the

schema for a user group or an application, as in the three-level schema architecture. All the problems related to query processing, transaction processing, and directory and metadata management and recovery apply to FDBSs with additional considerations. It is not within our scope to discuss them in detail here.

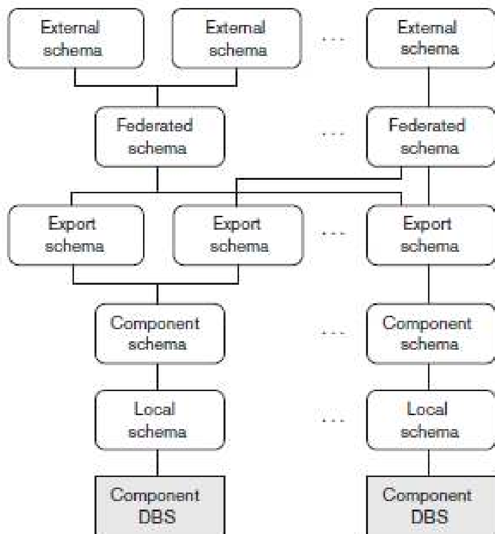


Figure 23.9
The five-level schema architecture in a federated database system (FDBS).

Source: Adapted from Sheth and Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases." *ACM Computing Surveys* (Vol. 22: No. 3, September 1990).

An Overview of Three-Tier Client/Server Architecture

As we pointed out in the chapter introduction, full-scale DDBMSs have not been developed to support all the types of functionalities that we have discussed so far. Instead, distributed database applications are being developed in the context of the client/server architectures. We introduced the two-tier client/server architecture in It is now more common to use a three-tier architecture rather than a two-tier architecture, particularly in Web applications. This architecture is illustrated in Figure 23.10. In the three-tier client/server architecture, the following three layers exist:

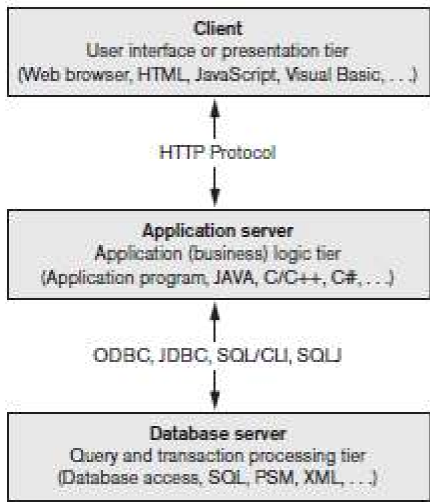


Figure 23.10
The three-tier client/server architecture.

1. **Presentation layer (client).** This provides the user interface and interacts with the user. The programs at this layer present Web interfaces or forms to the client in order to interface with the application. Web

browsers are often utilized, and the languages and specifications used include HTML, XHTML, CSS, Flash, MathML, Scalable Vector Graphics (SVG), Java, JavaScript, Adobe Flex, and others. This layer handles user input, output, and navigation by accepting user commands and displaying the needed information, usually in the form of static or dynamic Web pages. The latter are employed when the interaction involves database access. When a Web interface is used, this layer typically communicates with the application layer via the HTTP protocol.

2. Application layer (business logic). This layer programs the application logic. For example, queries can be formulated based on user input from the client, or query results can be formatted and sent to the client for presentation. Additional application functionality can be handled at this layer, such as security checks, identity verification, and other functions. The application layer can interact with one or more databases or data sources as needed by connecting to the database using ODBC, JDBC, SQL/CLI, or other database access techniques.

3. Database server. This layer handles query and update requests from the application layer, processes the requests, and sends the results. Usually SQL is used to access the database if it is relational or object-relational, and stored database procedures may also be invoked. Query results (and queries) may be formatted into XML (see Chapter 13) when transmitted between the application server and the database server. Exactly how to divide the DBMS functionality among the client, application server, and database server may vary. The common approach is to include the functionality of a centralized DBMS at the database server level. A number of relational DBMS products have taken this approach, in which an **SQL server** is provided. The application server must then formulate the appropriate SQL queries and connect to the database server when needed. The client provides the processing for user interface interactions. Since SQL is a relational standard, various SQL servers, possibly provided by different vendors, can accept SQL commands through standards such as ODBC, JDBC, and SQL/CLI (see Chapter 10). In this architecture, the application server may also refer to a data dictionary that includes information on the distribution of data among the various SQL servers, as well as modules for decomposing a global query into a number of local queries that can be executed at the various sites. Interaction between an application server and database server might proceed as follows during the processing of an SQL query:

1. The application server formulates a user query based on input from the client layer and decomposes it into a number of independent site queries. Each site query is sent to the appropriate database server site.
2. Each database server processes the local query and sends the results to the application server site. Increasingly, XML is being touted as the standard for data exchange (see Chapter 13), so the database server may format the query result into XML before sending it to the application server.
3. The application server combines the results of the subqueries to produce the result of the originally required query, formats it into HTML or some other form accepted by the client, and sends it to the client site for display. The application server is responsible for generating a distributed execution plan for a multisite query or transaction and for supervising distributed execution by sending commands to servers. These commands include local queries and transactions to be executed, as well as commands to transmit data to other clients or servers. Another function controlled by the application server (or coordinator) is that of ensuring consistency of replicated copies of a data item by employing distributed (or global) concurrency control techniques. The application server must also ensure the atomicity of global transactions by performing global recovery when certain sites fail.

If the DDBMS has the capability to *hide* the details of data distribution from the application server, then it enables the application server to execute global queries and transactions as though the database were centralized, without having to specify the sites at which the data referenced in the query or transaction

resides. This property is called **distribution transparency**. Some DDBMSs do not provide distribution transparency, instead requiring that applications are aware of the details of data distribution.

Distributed Database Concepts

We can define a **distributed database (DDB)** as a collection of multiple logically interrelated databases distributed over a computer network, and a **distributed database management system (DDBMS)** as a software system that manages a distributed database while making the distribution transparent to the user.

23.1.1 What Constitutes a DDB

For a database to be called distributed, the following minimum conditions should be satisfied:

■ **Connection of database nodes over a computer network.** There are multiple computers, called **sites** or **nodes**. These sites must be connected by an underlying **network** to transmit data and commands among sites.

■ **Logical interrelation of the connected databases.** It is essential that the information in the various database nodes be logically related.

■ **Possible absence of homogeneity among connected nodes.** It is not necessary that all nodes be identical in terms of data, hardware, and software. The sites may all be located in physical proximity—say, within the same building or a group of adjacent buildings—and connected via a **local area network**, or they may be geographically distributed over large distances and connected via a **long-haul** or **wide area network**. Local area networks typically use wireless hubs or cables, whereas long-haul networks use telephone lines, cables, wireless communication infrastructures, or satellites. It is common to have a combination of various types of networks.

Networks may have different **topologies** that define the direct communication paths among sites. The type and topology of the network used may have a significant impact on the performance and hence on the strategies for distributed query processing and distributed database design. For high-level architectural issues, however, it does not matter what type of network is used; what matters is that each site be able to communicate, directly or indirectly, with every other site. For the remainder of this chapter, we assume that some type of network exists among nodes, regardless of any particular topology. We will not address any networks pecific issues, although it is important to understand that for an efficient operation of a distributed database system (DDBS), network design and performance issues are critical and are an integral part of the overall solution. The details of the underlying network are invisible to the end user.

23.1.2 Transparency

The concept of transparency extends the general idea of hiding implementation details from end users. A highly transparent system offers a lot of flexibility to the end user/application developer since it requires little or no awareness of underlying details on their part. In the case of a traditional centralized database, transparency simply pertains to logical and physical data independence for application developers. However, in a DDB scenario, the data and software are distributed over multiple nodes connected by a computer network, so additional types of transparencies are introduced. Consider the company database in Figure 5.5

that we have been discussing throughout the book. The EMPLOYEE, PROJECT, and WORKS_ON tables may be fragmented horizontally (that is, into sets of rows, as we will discuss in Section 23.2) and stored with possible replication, as shown in Figure 23.1. The following types of transparencies are possible:

■ **Data organization transparency** (**also known as *distribution* or *network transparency***)**.** This refers to freedom for the user from the operational details of the network and the placement of the data in the distributed system. It may be divided into location transparency and naming transparency.

■ **Location transparency** refers to the fact that the command used to perform a task is independent of the location of the data and the location of the node where the command was issued. **Naming transparency** implies that once a name is associated with an object, the named objects can be accessed unambiguously without additional specification as to where the data is located.

■ **Replication transparency.** As we show in Figure 23.1, copies of the same data objects may be stored at multiple sites for better availability, performance, and reliability. Replication transparency makes the user unaware of the existence of these copies.

■ **Fragmentation transparency.** Two types of fragmentation are possible.

Horizontal fragmentation distributes a relation (table) into sub relations that are subsets of the tuples (rows) in the original relation; this is also known as **sharding** in the newer big data and cloud computing systems. **Vertical fragmentation** distributes a relation into sub relations where each sub relation is defined by a subset of the columns of the original relation. Fragmentation transparency makes the user unaware of the existence of fragments.

■ Other transparencies include **design transparency** and **execution transparency**—which refer, respectively, to freedom from knowing how the distributed database is designed and where a transaction executes.

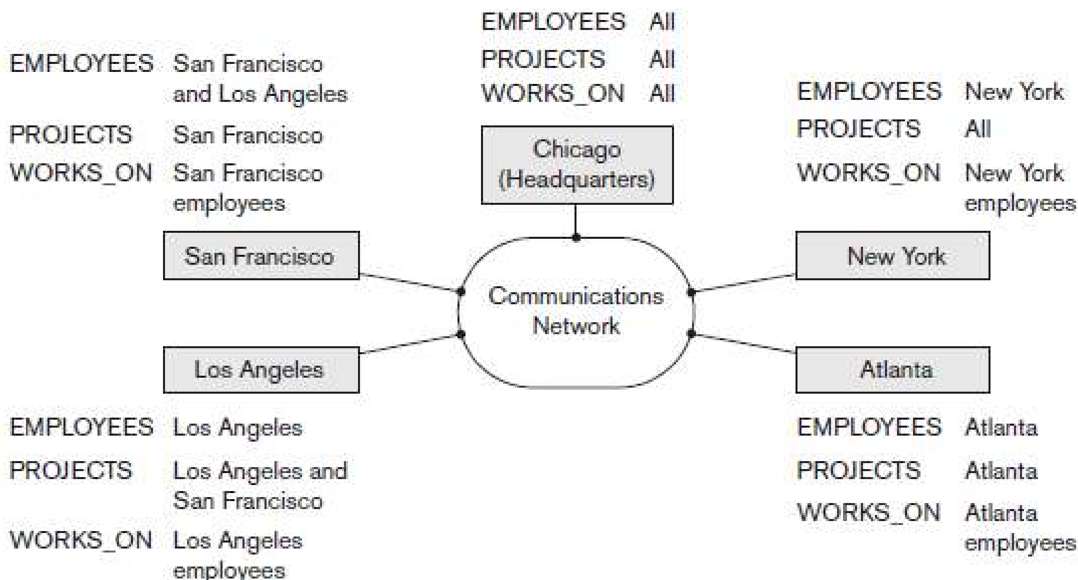


Figure 23.1
Data distribution and replication among distributed databases.

23.1.3 Availability and Reliability

Reliability and availability are two of the most common potential advantages cited for distributed databases. **Reliability** is broadly defined as the probability that a system is running (not down) at a certain time point, whereas **availability** is the probability that the system is continuously available during a time interval. We can directly relate reliability and availability of the database to the faults, errors, and failures associated with it. A failure can be described as a deviation of a system's behavior from that which is specified in order to ensure correct execution of operations.

Errors constitute that subset of system states that causes the failure. **Fault** is the cause of an error. To construct a system that is reliable, we can adopt several approaches. One common approach stresses *fault tolerance*; it recognizes that faults will occur, and it designs mechanisms that can detect and remove faults before they can result in a system failure. Another more stringent approach attempts to ensure that the final system does not contain any faults. This is done through an exhaustive design process followed by extensive quality control and testing. A reliable DDBMS tolerates failures of underlying components, and it processes user requests as long as database consistency is not violated. A DDBMS recovery manager has to deal with failures arising from transactions, hardware, and communication networks. Hardware failures can either be those that result in loss of main memory contents or loss of secondary storage contents. Network failures occur due to errors associated with messages and line failures. Message errors can include their loss, corruption, or out-of-order arrival at destination. The previous definitions are used in computer systems in general, where there is a technical distinction between reliability and availability. In most discussions related to DDB, the term **availability** is used generally as an umbrella term to cover both concepts.

23.1.4 Scalability and Partition Tolerance

Scalability determines the extent to which the system can expand its capacity while continuing to operate without interruption. There are two types of scalability:

1. **Horizontal scalability:** This refers to expanding the number of nodes in the distributed system. As nodes are added to the system, it should be possible to distribute some of the data and processing loads from existing nodes to the new nodes.

2. **Vertical scalability:** This refers to expanding the capacity of the individual nodes in the system, such as expanding the storage capacity or the processing power of a node. As the system expands its number of nodes, it is possible that the network, which connects the nodes, may have faults that cause the nodes to be partitioned into groups of nodes. The nodes within each partition are still connected by a subnetwork, but communication among the partitions is lost. The concept of **partition tolerance** states that the system should have the capacity to continue operating while the network is partitioned.

23.1.5 Autonomy

Autonomy determines the extent to which individual nodes or DBs in a connected DDB can operate independently. A high degree of autonomy is desirable for increased flexibility and customized maintenance of an individual node. Autonomy can be applied to design, communication, and execution. **Design autonomy** refers to independence of data model usage and transaction management techniques among nodes. **Communication autonomy** determines the extent to which each node can decide on sharing of information with other nodes. **Execution autonomy** refers to independence of users to act as they please.

23.1.6 Advantages of Distributed Databases

Some important advantages of DDB are listed below.

1. **Improved ease and flexibility of application development**. Developing and maintaining applications at geographically distributed sites of an organization is facilitated due to transparency of data distribution and control.

2. **Increased availability.** This is achieved by the isolation of faults to their site of origin without affecting the other database nodes connected to the network. When the data and DDBMS software are distributed over many sites, one site may fail while other sites continue to operate. Only the data and software that exist at the failed site cannot be accessed. Further improvement is achieved by judiciously replicating data and software at more than one site. In a centralized system, failure at a single site makes the whole system unavailable to all users. In a distributed database, some of the data may be unreachable, but users may still be able to access other parts of the database. If the data in the failed site has been replicated at another site prior to the failure, then the user will not be affected at all. The ability of the system to survive network partitioning also contributes to high availability.

3. **Improved performance.** A distributed DBMS fragments the database by keeping the data closer to where it is needed most. **Data localization** reduces the contention for CPU and I/O services and simultaneously reduces access delays involved in wide area networks. When a large database is distributed over multiple sites, smaller databases exist at each site. As a result, local queries and transactions accessing data at a single site have better performance because of the smaller local databases. In addition, each site has a smaller number of transactions executing than if all transactions are submitted to a single centralized database. Moreover, interquery and intraquery parallelism can be achieved by executing multiple queries at different sites, or by breaking up a query into a number of subqueries that execute in parallel. This contributes to improved performance.

4. **Easier expansion via scalability**. In a distributed environment, expansion of the system in terms of adding more data, increasing database sizes, or adding more nodes is much easier than in centralized (non-distributed) systems. The transparencies we discussed in Section 23.1.2 lead to a compromise between ease of use and the overhead cost of providing transparency. Total transparency provides the global user with a view of the entire DDBS as if it is a single centralized system. Transparency is provided as a complement to **autonomy**, which gives the users tighter control over local databases. Transparency features may be implemented as a part of the user language, which may translate the required services into appropriate operations.

Distributed Data Storage

Assume relational data model

_ Replication: system maintains multiple copies of data, stored in different sites, for faster retrieval and fault tolerance.

_ Fragmentation: relation is partitioned into several fragments stored in distinct sites.

_ Replication and fragmentation: relation is partitioned into several fragments; system maintains several identical replicas of each such fragment.

Data Replication

_ A relation or fragment of a relation is replicated if it is stored redundantly in two or more sites.

_ Full replication of a relation is the case where the relation is

stored at all sites.
_ Fully redundant databases are those in which every site contains a copy of the entire database.

Advantages of Replication

– Availability: failure of a site containing relation r does not result in unavailability of r if replicas exist.

– Parallelism: queries on r may be processed by several nodes in parallel.

– Reduced data transfer: relation r is available locally at each site containing a replica of r .

Disadvantages of Replication

– Increased cost of updates: each replica of relation r must be updated.

– Increased complexity of concurrency control: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented.

Data Fragmentation

_ Division of relation r into fragments r1, r2, ..., rn which contain sufficient information to reconstruct relation r .

_ Horizontal fragmentation: each tuple of r is assigned to one or more fragments.

_ Vertical fragmentation: the schema for relation r is split into several smaller schemas.

– All schemas must contain a common candidate key (or superkey) to ensure lossless join property.

– A special attribute, the tuple-id attribute may be added to each schema to serve as a candidate key.

_ Fragments may be successively fragmented to an arbitrary depth. Vertical and horizontal fragmentation can be mixed.

Example: relation account with following schema
Account-schema = (branch-name, account-number, balance

Horizontal Fragmentation of *account* Relation

branch-name	*account-number*	*balance*
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

$account_1$

branch-name	*account-number*	*balance*
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

Vertical Fragmentation of *deposit* Relation

branch-name	*customer-name*	*tuple-id*
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

$deposit_1$

account-number	*balance*	*tuple-id*
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

$deposit_2$

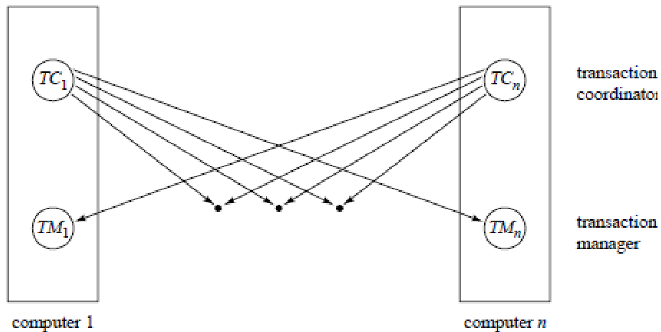
Advantages of Fragmentation

- Horizontal:
  - allows parallel processing on a relation
  - allows a global table to be split so that tuples are located where they are most frequently accessed
- Vertical:
  - allows for further decomposition than can be achieved with normalization
  - tuple-id attribute allows efficient joining of vertical fragments
  - allows parallel processing on a relation
  - allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed

Distributed Transaction Model

- Transactions may access data at several sites
- Each site has a *local transaction manager* responsible for:
  - Maintaining a log for recovery purposes.
  - Participating in coordinating the concurrent execution of th transactions executing at that site.
- Each site has a *transaction coordinator*, which is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing subtransactions to appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites.

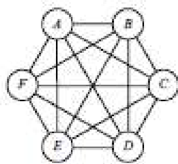
Transaction System Architecture



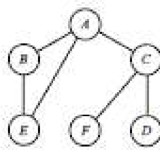
System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of messages.
  - Failure of a communication link.
  - Network partition.
- The configurations of how sites are connected physically can be compared in terms of:
  - Installation cost.
  - Communication cost.
  - Availability.

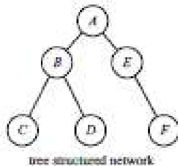
Network Topology



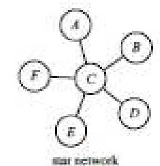
fully connected network



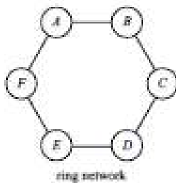
partially connected network



tree structured network



star network



ring network

System Failure Modes (Cont.)

- Partially connected networks have direct links between some, but not all, pairs of sites.
  - Lower installation cost than fully connected network
  - Higher communication cost to *route* messages between two sites that are not directly connected

Network Topology (Cont.)

- A *partitioned* system is split into two (or more) subsystems (*partitions*) that lack any connection.
- Tree-structured: low installation and communication costs; the failure of a single link can partition network
- Ring: At least two links must fail for partition to occur; communication cost is high
- Star:
  - the failure of a single link results in a network partition, but since one of the partitions has only a single site it can be treated as a single-site failure
  - low communication cost
  - failure of the central site results in every site in the system becoming disconnected

Robustness

- A robust system must:
  - Detect site or link failures
  - Reconfigure the system so that computation may continue.
  - Recover when a processor or link is repaired.
- Handling failure types:
  - Retransmit lost messages.
  - Unacknowledged retransmits indicate link failure; find alternative route for message.
  - Failure to find alternative route is a symptom of network partition.
- Network link failures and site failures are generally indistinguishable.

Procedure to Reconfigure System

- If replicated data is stored at the failed site, update the catalog so that queries do not reference the copy at the failed site.
- Transactions active at the failed site should be aborted.
- If the failed site is a central server for some subsystem, an *election* must be held to determine the new server.
- Reconfiguration scheme must work correctly in case of network partitioning; must avoid:
  - Electing two or more central servers in distinct partitions.
  - Updating replicated data item by more than one partition
- Represent recovery tasks as a series of transactions; concurrent control subsystem and transaction management subsystem may then be relied upon for proper reintegration.

Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  - not acceptable to have a transaction committed at one site and aborted at another
- The *two-phase commit* (2PC) protocol is widely used — will study this first
- The *three-phase commit* (3PC) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol.

Two-Phase Commit Protocol (2PC)

- Assumes *fail-stop* model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached
- The protocol involves all the local sites at which the transaction executed
- Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i.

Phase 1: Obtaining a Decision

- Coordinator asks all participants to *prepare* to commit transaction T_i
  - C_i adds the record <**prepare** T> to the log and forces log to stable storage
  - sends **prepare** T message to all sites at which T executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <**no** T> to the log and send **abort** T message to C_i
  - if the transaction can be committed, then:
    - add the record <**ready** T> to the log
    - force *all log records* for T to stable storage
    - send **ready** T message to C_i

Phase 2: Recording the Decision

- T can be committed if C_i received a **ready** T message from al the participating sites; otherwise T must be aborted
- Coordinator adds a decision record, <**commit** T> or <**abort** T>, to the log and forces record onto stable storage Once that record reaches stable storage it is irrevocable (ever if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally

Handling of Failures – Site Failure

When site S_k recovers, it examines its log to determine the fate of transactions active at the time of the failure

- Log contains <**commit** *T*> record: site executes **redo**(*T*).
- Log contains <**abort** *T*> record: site executes **undo**(*T*).
- Log contains <**ready** *T*> record: site must consult C_i to determine the fate of *T*.
  - if *T* committed, **redo**(*T*)
  - if *T* aborted, **undo**(*T*)
- The log contains no control records concerning *T*: implies that S_k failed before responding to the **prepare** *T* message from C_i
  - since the failure of S_k precludes the sending of such a response, C_i must abort *T*
  - S_k must execute **undo**(*T*)

Handling of Failures – Coordinator Failure

If coordinator fails while the commit protocol for *T* is executing, then participating sites must decide on *T*'s fate:

- If an active site contains a <**commit** *T*> record in its log, then *T* must be committed.
- If an active site contains an <**abort** *T*> record in its log, then *T* must be aborted.
- If some active site does *not* contain a <**ready** *T*> record in its log, then the failed coordinator C_i cannot have decided to commit *T*. Can therefore abort *T*.
- If none of the above cases holds, then all active sites must have a <**ready** *T*> record in their logs, but no additional control records (such as <**abort** *T*> or <**commit** *T*>). In this case active sites must wait for C_i to recover, to find decision.
- *Blocking* problem: active sites may have to wait for failed coordinator to recover.

Handling of Failures – Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to wait for decision from coordinator
  - The coordinator and the sites that are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
    - Again, no harm results

Recovery and Concurrency Control

- *In-doubt* transactions have a <**ready** *T*>, but neither a <**commit** *T*>, nor an <**abort** *T*> log record.
- The recovering site must determine the commit–abort status of such transactions by contacting other sites; this can slow and potentially block recovery.
- Recovery algorithms can note lock information in the log.
  - Instead of <**ready** *T*>, write out <**ready** *T*, *L*>
    L = list of locks held by *T* when the log is written (read locks can be omitted).
  - For every in-doubt transaction *T*, all the locks noted in the <**ready** *T*, *L*> log record are reacquired.
- After lock reacquisition, transaction processing can resume; the commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions.

Three Phase Commit (3PC)

- Assumptions:
  - No network partitioning
  - At any point, at least one site must be up.
  - At most *K* sites (participants as well as coordinator) can fail
- Phase 1: Obtaining Preliminary Decision: Identical to 2PC Phase 1.
  - Every site is ready to commit if instructed to do so
  - Under 2PC each site is obligated to wait for decision from coordinator
  - Under 3PC, knowledge of pre-commit decision can be used to commit despite coordinator failure

Phase 2: Recording the Preliminary Decision

_ Coordinator adds a decision record (<**abort** T> or <**precommit** T>) in its log and forces record to stable storage

_ Coordinator sends a message to each participant informing it of the decision

_ Participant records decision in its log

– If abort decision reached then participant aborts locally

– If pre-commit decision reached then participant replies with <**acknowledge** T>

Phase 3 : Recording Decision in the Database

Executed only if decision in phase 2 was to precommit

_ Coordinator collects acknowledgments. It sends <**commit** T> message to the participants as soon as it receives K acknowledgments.

_ Coordinator adds the record <**commit** T> in its log and forces record to stable storage.

_ Coordinator sends a message to each participant to <**commit** T>.

_ Participants take appropriate action locally

Handling Site Failure

Site Failure. Upon recovery, a participating site examines its log and does the following:

- Log contains <**commit** T> record: site executes **redo**(T).
- Log contains <**abort** T> record: site executes **undo**(T).
- Log contains <**ready** T> record, but no <**abort** T> or <**precommit** T> record: site consults C_i to determine the fate of T.
  - if C_i says T aborted, site executes **undo**(T) (and writes <**abort** T> record)
  - if C_i says T committed, site executes **redo**(T) (and writes <**commit** T> record)
  - if C_i says T precommitted, site resumes the protocol from receipt of **precommit** T message (thus recording <**precommit** T> in the log, and sending **acknowledge** T message sent to coordinator)

Handling Site Failure (Cont.)

- Log contains <**precommit** T> record, but no <**abort** T> or <**commit** T> record: site consults C_i to determine the fate of transaction T.
  - if C_i says T aborted, site executes **undo**(T)
  - if C_i says T committed, site executes **redo**(T)
  - if C_i says T still in precommit state, site resumes protocol at this point
- Log contains no <**ready** T> record for a transaction T: site executes **undo**(T) and writes <**abort** T> record.

Coordinator-Failure Protocol

1. The active participating sites select a new coordinator, C_{new}
2. C_{new} requests local status of T from each participating site
3. Each participating site, including C_{new}, determines the local status of T:
   - **Committed**. The log contains a <**commit** T> record.
   - **Aborted**. The log contains an <**abort** T> record.
   - **Ready**. The log contains a <**ready** T> record but no <**abort** T> or <**precommit** T> record.
   - **Precommitted**. The log contains a <**precommit** T> record but no <**abort** T> or <**commit** T> record.
   - **Not ready**. The log contains neither a <**ready** T> nor an <**abort** T> record.

   A site that failed and recovered must ignore any **precommit** record in its log when determining its status.
4. Each participating site sends its local status to C_{new}.

Coordinator Failure Protocol (Cont.)

5. C_{new} decides either to commit or abort T, or to restart the three-phase commit protocol:
   - Commit state for any one participant $\Rightarrow$ commit
   - Abort state for any one participant $\Rightarrow$ abort
   - Precommit state for any one participant and above 2 cases do not hold $\Rightarrow$
     A precommit message is sent to those participants in the uncertain state. Protocol is resumed from that point.
   - Uncertain state at all live participants $\Rightarrow$ abort
     Since at least $n - k$ sites are up, the fact that all participants are in an uncertain state means that the coordinator has not sent a <**commit** T> message, implying that no site has committed T.

Concurrency Control

- Modify concurrency control schemes for use in distributed environment.
- We assume that each site participates in the execution of a commit protocol to ensure global transaction atomicity.

Single-Lock-Manager Approach

- System maintains a *single* lock manager that resides in a *single* chosen site, say S_i.
- When a transaction needs to lock a data item, it sends a lock request to S_i and lock manager determines whether the lock can be granted immediately
  - If yes, lock manager sends a message to the site which initiated the request
  - If no, request is delayed until it can be granted, at which time a message is sent to the initiating site

Single-Lock-Manager Approach (Cont.)

- The transaction can read the data item from *any* one of the sites at which a replica of the data item resides.
- In the case of a write, all the sites where a replica of the data item resides must be involved in the writing.
- Advantages of scheme:
  - Simple implementation
  - Simple deadlock handling
- Disadvantages of scheme are:
  - Bottleneck: lock manager site becomes a bottleneck
  - Vulnerability: system is vulnerable to lock manager site failure

Majority Protocol

- Local lock manager at each site administers lock and unlock requests for data items stored at that site.
- When a transaction wishes to lock an unreplicated data item Q residing at site S_i, a message is sent to S_i's lock manager.
- If Q is locked in an incompatible mode, then the request is delayed until it can be granted.
- When the lock request can be granted, the lock manager sends a message back to the initiator indicating that the lock request has been granted.
- Advantage of simple implementation, however, since lock and unlock requests are no longer made at a single site, deadlock handling is more complex.

Majority Protocol (Cont.)

- In case of replicated data, majority protocol is more complicated to implement than the previous schemes
  - If Q is replicated at n sites, then a lock request message must be sent to more than half of the n sites in which Q is stored.
  - The transaction does not operate on Q until it has obtained a lock on a majority of the replicas of Q.
  - When writing the data item, transaction performs writes on all replicas.
- Requires $2(n/2 + 1)$ messages for handling lock requests, and $(n/2 + 1)$ messages for handling unlock requests.
- Potential for deadlock even with single item — e.g., each of 3 transactions may have locks on 1/3rd of the replicas of a data item

Biased Protocol

- Local lock manager at each site as in majority protocol, however, requests for shared locks are handled differently than requests for exclusive locks.
  - **Shared locks**. When a transaction needs to lock data item Q, it simply requests a lock on Q from the lock manager at one site containing a replica of Q.
  - **Exclusive locks**. When a transaction needs to lock data item Q, it requests a lock on Q from the lock manager at all sites containing a replica of Q.
- Advantage — imposes less overhead on **read** operations.
- Disadvantage — additional overhead on writes and complexity in handling deadlock.

Primary Copy

- Choose one replica to be the primary copy, which must reside in precisely one site (e.g., *primary site of Q*).
- When a transaction needs to lock a data item Q, it requests a lock at the primary site of Q.
- Concurrency control for replicated data handled similarly to unreplicated data—simple implementation.
- If the primary site of Q fails, Q is inaccessible even though other sites containing a replica may be accessible.

Timestamping

- Each site generates a unique local timestamp using either a logical counter or the local clock.
- Global unique timestamp is obtained by concatenating the unique local timestamp with the unique site identifier.

locally-unique timestamp	globally-unique site identifier

Timestamping (Cont.)

A site with a slow clock will assign smaller timestamps → "disadvantages" transactions

Define within each site S_i a *logical clock* (LC_i), which generates the unique local timestamp

Require that S_i advance its logical clock whenever a transaction T_i with timestamp $<x,y>$ visits that site and x is greater than the current value of LC_i.

In this case, site S_i advances its logical clock to the value $x + 1$.

Distributed Query Processing

- For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.
- In a distributed system, other issues must be taken into account:
  - The cost of data transmission over the network.
  - The potential gain in performance from having several sites process parts of the query in parallel.

Query Transformation

Translating algebraic queries to queries on fragments.

- It must be possible to construct relation *r* from its fragments
- Replace relation *r* by the expression to construct relation *r* from its fragments

Site selection for query processing.

Example Query

- Consider the horizontal fragmentation of the *account* relation into

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

- The query $\sigma_{branch\text{-}name = \text{“Hillside”}}(account)$ becomes

$$\sigma_{branch\text{-}name = \text{“Hillside”}}(account_1 \cup account_2)$$

which is optimized into

$$\sigma_{branch\text{-}name = \text{“Hillside”}}(account_1) \cup \sigma_{branch\text{-}name = \text{“Hillside”}}(account_2)$$

Example Query (Cont.)

- Since $account_1$ has only tuples pertaining to the Hillside branch, we can eliminate the selection operation.
- Apply the definition of $account_2$ to obtain

$$\sigma_{branch\text{-}name = \text{“Hillside”}}(\sigma_{branch\text{-}name = \text{“Valleyview”}}(account))$$

- This expression is the empty set regardless of the contents of the *account* relation.
- Final strategy is for the Hillside site to return $account_1$ as the result of the query.

Simple Join Processing

Consider the following relational algebra expression in which the three relations are neither replicated nor fragmented

$$account \bowtie depositor \bowtie branch$$

- *account* is stored at site S_1
- *depositor* at S_2
- *branch* at S_3
- For a query issued at site S_I, the system needs to produce the result at site S_I.

Possible Query Processing Strategies

- Ship copies of all three relations to site S_I and choose a strategy for processing the entire query locally at site S_I.
- Ship a copy of the *account* relation to site S_2 and compute $temp_1 = account \bowtie depositor$ at S_2. Ship $temp_1$ from S_2 to S_3, and compute $temp_2 = temp_1 \bowtie branch$ at S_3. Ship the result $temp_2$ to S_I.
- Devise similar strategies, exchanging the roles of S_1, S_2, S_3.
- Must consider following factors:
  - amount of data being shipped
  - cost of transmitting a data block between sites
  - relative processing speed at each site

Semijoin Strategy

- Let r_1 be a relation with schema R_1 stored at site S_1
  Let r_2 be a relation with schema R_2 stored at site S_2
- Evaluate the expression $r_1 \bowtie r_2$, and obtain the result at S_1.

1. Compute $temp1 \leftarrow \Pi_{R_1 \cap R_2}(r_1)$ at S_1.
2. Ship $temp1$ from S_1 to S_2.
3. Compute $temp2 \leftarrow r_2 \bowtie temp1$ at S_2.
4. Ship $temp2$ from S_2 to S_1.
5. Compute $r_1 \bowtie temp2$ at S_1. This is the result of $r_1 \bowtie r_2$.

Formal Definition

- The semijoin of r_1 with r_2, is denoted by:

$$r_1 \rhd\!< r_2$$

it is defined by:

$$\Pi_{R_1}(r_1 \bowtie r_2)$$

- Thus, $r_1 \rhd\!< r_2$ selects those tuples of r_1 that contributed to $r_1 \bowtie r_2$.
- In step 3 above, $temp2 = r_2 \rhd\!< r_1$.
- For joins of several relations, the above strategy can be extended to a series of semijoin steps.

Join Strategies that Exploit Parallelism

- Consider $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$ where relation r_i is stored at site S_i. The result must be presented at site S_1.
- Pipelined-join strategy
  - r_1 is to S_2 and $r_1 \bowtie r_2$ is computed at S_2; simultaneously r_3 is shipped to S_4 and $r_3 \bowtie r_4$ is computed at S_4
  - S_2 ships tuples of $(r_1 \bowtie r_2)$ to S_1 as they are produced; S_4 ships tuples of $(r_3 \bowtie r_4)$ to S_1
  - Once tuples of $(r_1 \bowtie r_2)$ and $(r_3 \bowtie r_4)$ arrive at S_1, $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$ is computed in parallel with the computation of $(r_1 \bowtie r_2)$ at S_2 and the computation of $(r_3 \bowtie r_4)$ at S_4.

UNIT II SPATIAL AND TEMPORAL DATABASES

Active Databases Model – Design and Implementation Issues - Temporal Databases - Temporal Querying - Spatial Databases: Spatial Data Types, Spatial Operators and Queries – Spatial Indexing and Mining – Applications -– Mobile Databases: Location and Handoff Management, Mobile Transaction Models – Deductive Databases - Multimedia Databases.

2.1 Active Databases Model

A trigger is a procedure which is automatically invoked by the DBMS in response to changes to the database, and is specified by the database administrator (DBA). A database with a set of associated triggers is generally called an active database.

Parts of trigger

A triggers description contains three parts, which are as follows

1. The **event(s)** that triggers the rule: These events are usually database update operations that are explicitly applied to the database. However, in the general model, they could also be temporal events[2] or other kinds of external events

2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated. If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed

3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed.

Use of trigger

Triggers may be used for any of the following reasons −

- To implement any complex business rule, that cannot be implemented using integrity constraints.
- Triggers will be used to audit the process. For example, to keep track of changes made to a table.
- Trigger is used to perform automatic action when another concerned action takes place.

Types of triggers

The different types of triggers are explained below −

- **Statement level trigger** − It is fired only once for DML statement irrespective of number of rows affected by statement. Statement-level triggers are the default type of trigger.
- **Before-triggers** − At the time of defining a trigger we can specify whether the trigger is to be fired before a command like INSERT, DELETE, or UPDATE is executed or after the command is executed. Before triggers are automatically used to check the validity of data before the action is performed. For instance, we can use before trigger to prevent deletion of rows if deletion should not be allowed in a given case.

- **After-triggers** − It is used after the triggering action is completed. For example, if the trigger is associated with the INSERT command then it is fired after the row is inserted into the table.
- **Row-level triggers** − It is fired for each row that is affected by DML command. For example, if an UPDATE command updates 150 rows then a row-level trigger is fired 150 times whereas a statement-level trigger is fired only for once.

Create database trigger

To create a database trigger, we use the CREATE TRIGGER command. The details to be given at the time of creating a trigger are as follows −

- Name of the trigger.
- Table to be associated with.
- When trigger is to be fired: before or after.
- Command that invokes the trigger- UPDATE, DELETE, or INSERT.
- Whether row-level triggers or not.
- Condition to filter rows.
- PL/SQL block is to be executed when trigger is fired.

The syntax to create database trigger is as follows −

```
CREATE [OR REPLACE] TRIGGER triggername
{BEFORE|AFTER}
{DELETE|INSERT|UPDATE[OF COLUMNS]} ON table
[FOR EACH ROW {WHEN condition]]
[REFERENCE [OLD AS old] [NEW AS new]]
BEGIN
PL/SQL BLOCK
END.
```

2.2 Design and Implementation Issues for Active Databases

In this section, we discuss some additional issues concerning how rules are designed and implemented. The first issue concerns activation, deactivation, and grouping of rules. In addition to creating rules, an active database system should allow users to activate, deactivate, and drop rules by referring to their rule names. A deactivated rule will not be triggered by the triggering event.

This feature allows users to selectively deactivate rules for certain periods of time when they are not needed. The activate command will make the rule active again. The drop command deletes the rule from the system. Another option is to group rules into named rule sets, so the whole set of rules can be activated, deactivated, or dropped. It is also useful to have a command that can trigger a rule or rule set via an explicit PROCESS RULES command issued by the user.

The second issue concerns whether the triggered action should be executed before, after, instead of, or concurrently with the triggering event. A before trigger executes the trigger before executing the event that caused the trigger. It can be used in appli- cations such as checking for

constraint violations. An after trigger executes the trig- ger after executing the event, and it can be used in applications such as maintaining derived data and monitoring for specific events and conditions. An instead of trigger executes the trigger instead of executing the event, and it can be used in applications such as executing corresponding updates on base relations in response to an event that is an update of a view.

A related issue is whether the action being executed should be considered as a separate transaction or whether it should be part of the same transaction that triggered the rule. We will try to categorize the various options. It is important to note that not all options may be available for a particular active database system. In fact, most com- mercial systems are limited to one or two of the options that we will now discuss.

Let us assume that the triggering event occurs as part of a transaction execution. We should first consider the various options for how the triggering event is related to the evaluation of the rule's condition.

The rule condition evaluation is also known as rule consideration , since the action is to be executed only after considering whether the condition evaluates to true or false. There are three main possibilities for rule consideration:

1.Immediate consideration. The condition is evaluated as part of the same transaction as the triggering event, and is evaluated immediately.

This case can be further categorized into three options:

Evaluate the condition before executing the triggering event.

Evaluate the condition after executing the triggering event.

Evaluate the condition instead of executing the triggering event.

2. Deferred consideration. The condition is evaluated at the end of the trans- action that included the triggering event. In this case, there could be many

3. Detached consideration. The condition is evaluated as a separate transaction, spawned from the triggering transaction.

The next set of options concerns the relationship between evaluating the rule condition and executing the rule action. Here, again, three options are possible: immediate , deferred, or detached execution. Most active systems use the first option.

That is, as soon as the condition is evaluated, if it returns true, the action is immediately executed.

The Oracle system uses the immediate consideration model, but it allows the user to specify for each rule whether the before or after option is to be used with immediate condition evaluation. It also uses the immediate execution model. The STARBURST system uses the deferred consideration option, meaning that all rules triggered by a transaction wait until the triggering transaction reaches its end and issues its COMMIT WORK command before the rule conditions are evaluated.

Another issue concerning active database rules is the distinction between row-level rules and statement-level rules. Because SQL update statements (which act as triggering events) can specify a set of tuples, one has to distinguish between whether the rule should be considered once for the whole statement or whether it should be considered separately for each row (that is, tuple) affected by the statement.

2.3 Temporal Database

A Temporal Database is a database with **built-in support for handling time sensitive data**. Usually, databases store information only about current state, and not about past states. For example in a employee database if the address or salary of a particular person changes, the database gets updated, the old value is no longer there. However for many applications, it is important to maintain the past or historical values and the time at which the data was updated. That is, the knowledge of evolution is required. That is where temporal databases are useful. It stores information about the past, present and future. Any data that is time dependent is called the temporal data and these are stored in temporal databases.
Temporal Databases store information about states of the real world across time. Temporal Database is a database with built-in support for handling data involving time. It stores information relating to past, present and future time of all events.

Examples Of Temporal Databases

- **Healthcare Systems**: Doctors need the patients' health history for proper diagnosis. Information like the time a vaccination was given or the exact time when fever goes high etc.
- **Insurance Systems**: Information about claims, accident history, time when policies are in effect needs to be maintained.
- **Reservation Systems**: Date and time of all reservations is important.

Temporal Aspects

There are two different aspects of time in temporal databases.

- **Valid Time**: Time period during which a fact is true in real world, provided to the system.
- **Transaction Time**: Time period during which a fact is stored in the database, based on transaction serialization order and is the timestamp generated automatically by the system.

Temporal Relation

Temporal Relation is one where each tuple has associated time; either valid time or transaction time or both associated with it.

- **Uni-Temporal Relations:** Has one axis of time, either Valid Time or Transaction Time.
- **Bi-Temporal Relations:** Has both axis of time – Valid time and Transaction time. It includes Valid Start Time, Valid End Time, Transaction Start Time, Transaction End Time.

Valid Time Example

Now let's see an example of a person, John:

- John was born on April 3, 1992 in Chennai.
- His father registered his birth after three days on April 6, 1992.

- John did his entire schooling and college in Chennai.
- He got a job in Mumbai and shifted to Mumbai on June 21, 2015.
- He registered his change of address only on Jan 10, 2016.

John's Data In Non-Temporal Database

In a non-temporal database, John's address is entered as Chennai from 1992. When he registers his new address in 2016, the database gets updated and the address field now shows his Mumbai address. The previous Chennai address details will not be available. So, it will be difficult to find out exactly when he was living in Chennai and when he moved to Mumbai.

te	Real world event	Address
ril 3, 1992	John is born	
ril 6, 1992	John's father registered his birth	Chennai
ne 21, 2015	John gets a job	Chennai
10, 2016	John registers his new address	Mumbai

Table:Non temporal Database

Uni-Temporal Relation (Adding Valid Time To John's Data)

To make the above example a temporal database, we'll be adding the time aspect also to the database. First let's add the valid time which is the time for which a fact is true in real world. Valid time is the time for which a fact is true in the real world. A valid time period may be in the past, span the current time, or occur in the future.

The valid time temporal database contents look like this:
Name, City, Valid From, Valid Till

In our example, john was born on 3rd April 1992. Even though his father registered his birth three days later, the valid time entry would be 3rd April of 1992. There are two entries for the valid time. The **Valid Start Time** and the **Valid End Time**. So in this case 3rd April 1992 is the valid start time. Since we do not know the valid end time we add it as infinity.
Johns father registers his birth on 6th April 1992, a new database entry is made:
Person(John, Chennai, 3-Apr-1992, ∞).

Similarly John changes his address to Mumbai on 10th Jan 2016. However, he has been living in Mumbai from 21st June of the previous year. So his valid time entry would be 21 June 2015.

On January 10, 2016 John reports his new address in Mumbai:
Person(John, Mumbai, 21-June-2015, ∞).

The original entry is updated.
Person(John, Chennai, 3-Apr-1992, 20-June-2015).
The table will look something like this with two additional entries:

Name	City	Valid From	Valid Till
John	Chennai	April 3, 1992	June 20, 2015
John	Mumbai	June 21, 2015	∞

Table:Uni-temporal Database

Bi-Temporal Relation (John's Data Using Both Valid And Transaction Time)
Next we'll see a bi-temporal database which includes both the valid time and transaction time. Transaction time records the time period during which a database entry is made. So, now the database will have four additional entries the **valid from**, **valid till**, **transaction entered** and **transaction superseded**.

The database contents look like this:
Name, City, Valid From, Valid Till, Entered, Superseded
First, when John's father records his birth the valid start time would be 3rd April 1992, his actual birth date. However, the transaction entered time would be 6th April 1992.

Johns father registers his birth on 6th April 1992:
Person(John, Chennai, 3-Apr-1992, ∞, 6-Apr-1992, ∞).
Similarly, when john registers his change of address in Mumbai, a new entry is made. The valid from time for this entry is 21st June 2015, the actual date from which he started living in Mumbai. whereas the transaction entered time would be 10th January 2016. We do not know how long he'll be living in Mumbai. So the transaction end time and the valid end time would be infinity. At the same time the original entry is updated with the valid till time and the transaction superseded time.

On January 10, 2016 John reports his new address in Mumbai:
Person(John, Mumbai, 21-June-2015, ∞, 10-Jan-2016, ∞).
The original entry is updated.
Person(John, Chennai, 3-Apr-1992, 20-June-2015, 6-Apr-1992, 10-Jan-2016).

Now the database looks something like this:

me	City	Valid From	Valid Till	Entered	Superseded

me	City	Valid From	Valid Till	Entered	Superseded
n	Chennai	April 3, 1992	June 20, 2015	April 6, 1992	Jan 10, 2016
n	Mumbai	June 21, 2015	∞	Jan 10, 2016	∞

Bi-temporal Database

Advantages

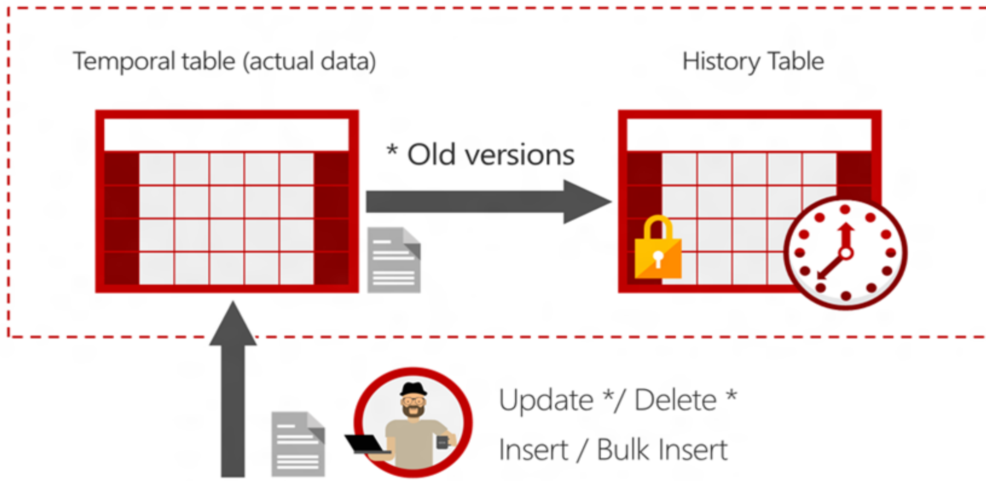
The main advantages of this bi-temporal relations is that it provides historical and roll back information. For example, you can get the result for a query on John's history, like: Where did John live in the year 2001?. The result for this query can be got with the valid time entry. The transaction time entry is important to get the rollback information.

- Historical Information – Valid Time.
- Rollback Information – Transaction Time.
- https://www.slideshare.net/janakiraman55/pamppt

Temporal Query

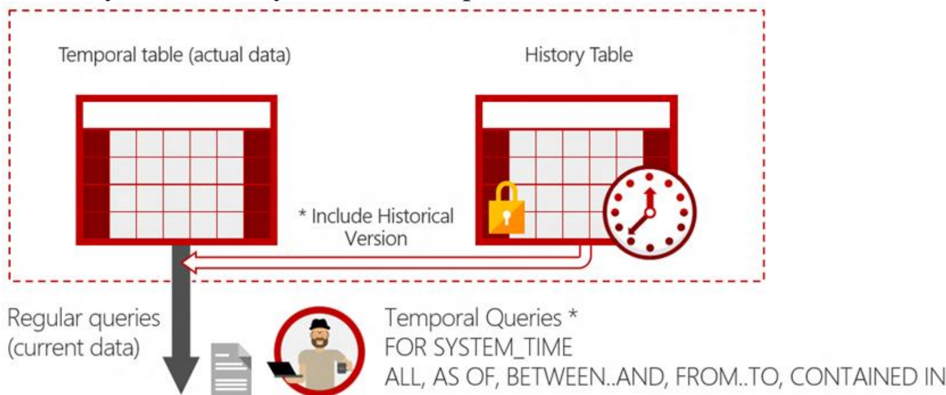
A TemporalQuery can be used to retrieve information from a temporal-based object.

When a temporal table is created in SQL Server, a history table is created behind the scenes. The main table contains the records as they exist at the current point in time and the history table contains all the previous versions of records. You can query the main table as normal or add temporal clauses to your query to find historical records.



Querying Temporal Tables

There are two main ways to query the history table. The first is looking at previous versions of the table by adding time-based clauses to your queries. The second is to look into the history table manually, which lets you see all the previous versions of records.



Creating a Temporal Table

First let's consider a normal table that we can use as our sample main table. For this post, we'll work with a basic person record:

```
create table dbo.Person
(
   [PersonId] int not null primary key clustered,
   [Name] nvarchar(max) not null,
   [Email] nvarchar(max) null,
   [Address] nvarchar(max) null,
   [PhoneNumber] nvarchar(max) null
)
```

That table is not temporal, but having the normal definition to compare it to will highlight what we need to make it temporal:

```
create table dbo.Person
(
   [PersonId] int not null primary key clustered identity(1,1),
   [Name] nvarchar(max) not null,
   [Email] nvarchar(max) null,
   [Address] nvarchar(max) null,
   [PhoneNumber] nvarchar(max) null,    [SysStartTime] datetime2 generated always as row start hidden
not null,
   [SysEndTime] datetime2 generated always as row end hidden not null,
   Period for system_time (SysStartTime, SysEndTime)
)
with (system_versioning = on (history_table = Person_History))
```

There are two main changes. First, we added SysStartTime and SysEndTime as generated columns then used that to create the Period column. These are required columns for temporal tables. Making the start and end time columns hidden is optional, but can help to hide the versioning when it's not needed. Second, we added with (system_versioning = on (…)) to the end of the statement. This will create the history table using the Period column defined above. The default naming for the history table is kind of messy, so we also defined what the table should be called. I like the _History suffix, so that's what we will use here.

2.5 Spatial database

Spatial data is associated with geographic locations such as cities,towns etc. A spatial database is optimized to store and query data representing objects. These are the objects which are defined in a geometric space.

A database is a collection of **related information** that permits the entry, storage, input, output, and organization of data. A database management system (DBMS) serves as an interface between users and their databases.

A ***spatial*** **database** includes **location**. It has geometry as points, lines, and polygons. GIS combines spatial data from many sources with many different people. Databases connect users to the GIS database.

For example, a city might have the wastewater division, land records, transportation, and fire departments connected and using datasets from common spatial databases. Let's take a closer look at spatial databases and how we use them in GIS.

Characteristics of Spatial Database

A spatial database system has the following characteristics

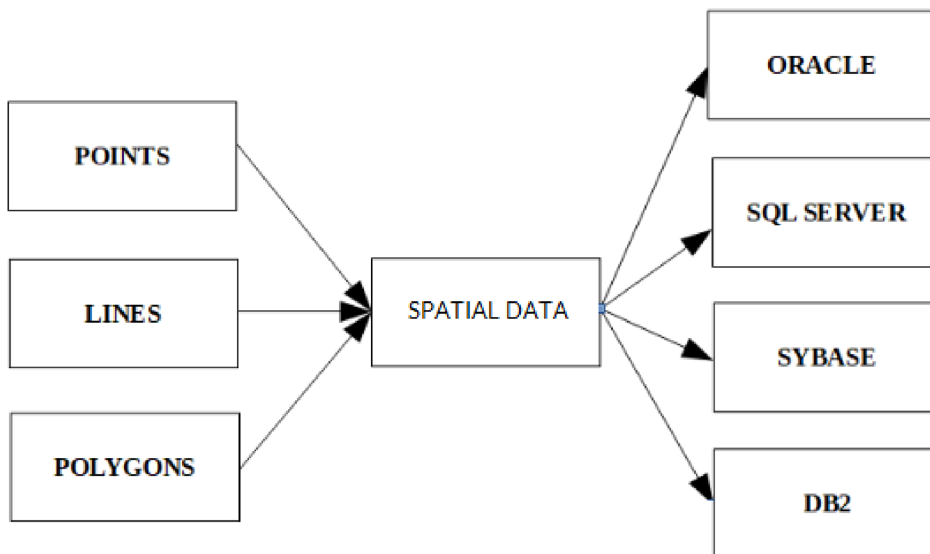
- It is a database system
- It offers spatial data types (SDTs) in its data model and query language.
- It supports spatial data types in its implementation, providing at least spatial indexing and efficient algorithms for spatial join.

Example

A road map is a visualization of geographic information. A road map is a 2-dimensional object which contains points, lines, and polygons that can represent cities, roads, and political boundaries such as states or provinces.

In general, spatial data can be of two types −

- Vector data: This data is represented as discrete points, lines and polygons
- Rastor data: This data is represented as a matrix of square cells.



The spatial data in the form of points, lines, polygons etc. is used by many different databases as shown above.

2.5.1 Spatial Data Types Overview

Spatial data represents information about the physical location and shape of geometric objects. These objects can be point locations or more complex objects such as countries, roads, or lakes.

SQL Server supports two spatial data types: the **geometry** data type and the **geography** data type.

- The **geometry** type represents data in a Euclidean (flat) coordinate system.
- The **geography** type represents data in a round-earth coordinate system.

Both data types are implemented as .NET common language runtime (CLR) data types in SQL Server.

There are two types of spatial data. The **geometry** data type supports planar, or Euclidean (flat-earth), data. The **geometry** data type both conforms to the *Open Geospatial Consortium (OGC) Simple Features for SQL Specification* version 1.1.0 and is compliant with SQL MM (ISO standard). SQL Server also supports the **geography** data type, which stores ellipsoidal (round-earth) data, such as GPS latitude and longitude coordinates.

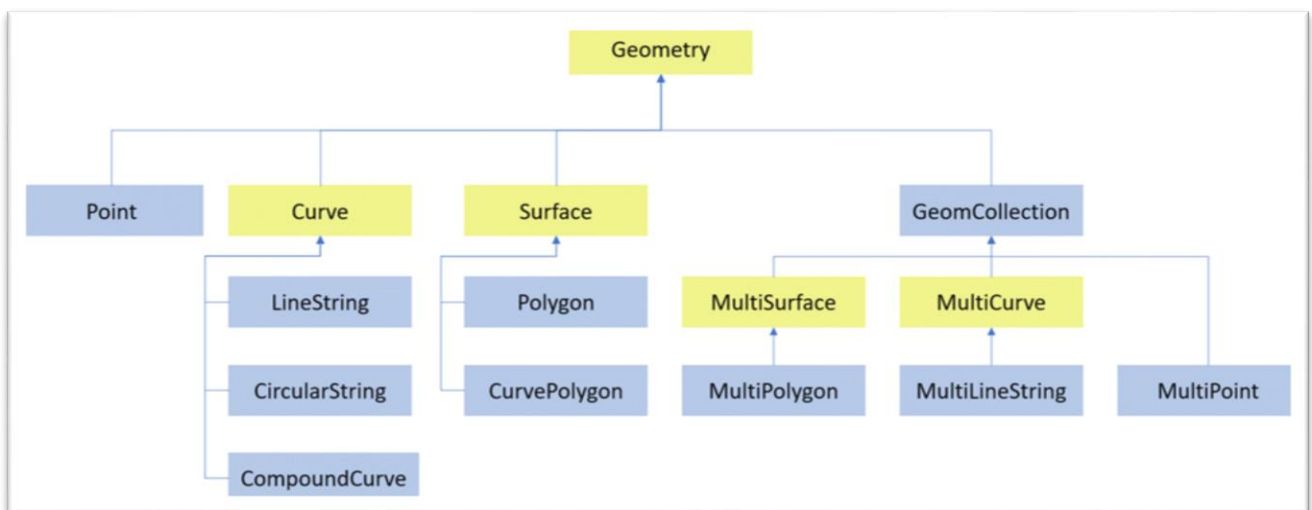
Tip

SQL Server spatial tools is a Microsoft sponsored open-source collection of tools for use with the spatial types in SQL Server. This project provides a set of reusable functions which applications can make use of. These functions may include data conversion routines, new transformations, aggregates, etc.

Spatial data objects

The **geometry** and **geography** data types support 16 types of spatial data objects, or instance types. However, only 11 of these instance types are *instantiable*; you can create and work with these instances (or instantiate them) in a database. These instances derive certain properties from their parent data types.

The figure below shows the geometry hierarchy upon which the **geometry** and **geography** data types are based. The instantiable types of **geometry** and **geography** are indicated in blue.



he hierarchy in Figure 1 includes:

- Data types for geographic features that can be perceived as forming a single unit; for example, individual residences and isolated lakes.
- Data types for geographic features that are made up of multiple units or components; for example, canal systems and groups of islands in a lake.
- A data type for geographic features of all kinds.

- **Data types for single-unit features**
  Use ST_Point, ST_LineString, and ST_Polygon to store coordinates that define the space occupied by features that can be perceived as forming a single unit.
- **Data types for multi-unit features**
  Use ST_MultiPoint, ST_MultiLineString, and ST_MultiPolygon to store coordinates that define spaces occupied by features that are made up of multiple units.
- **A data type for all features**
  You can use ST_Geometry when you are not sure which of the other data types to use.

The subtypes for geometry and geography types are divided into simple and collection types. Some methods like STNumCurves() work only with simple types.

Simple types are:

- Point
- LineString
- CircularString
- CompoundCurve
- Polygon
- CurvePolygon

Collection types are:

- MultiPoint
- MultiLineString
- MultiPolygon
- GeometryCollection

2.5.2 Spatial Operators and queries

1. Spatial operators :
Spatial operators these operators are applied in **geometric properties** of objects.
It is then used in the physical space to capture them and the relation among them.
It is also used to perform spatial analysis.
Spatial operators are grouped into three categories :

1. **Topological operators :**
   Topological properties do not vary when topological operations are applied, like translation or rotation.
   Topological operators are hierarchically structured in many levels. The base level offers operators, ability to check for detailed topological relations between regions with a broad boundary. The higher levels offer more abstract operators that allow users to query uncertain spatial data independent of the geometric data model.

Examples –
open (region), close (region), and inside (point, loop).

2. **Projective operators :**
Projective operators, like convex hull are used to establish predicates regarding the concavity convexity of objects.
    - Convexity is a measure of the curvature, or the degree of the curve, in the relationship between bond prices and bond yields.
    - Concavity relates to the rate of change of a function's derivative. A function *ff* is **concave up** (or upwards) where the derivative f′*f*′f, prime is increasing.

Example –
Having inside the object's concavity.

3. **Metric operators :**
Metric operators's task is to provide a more accurate description of the geometry of the object. They are often used to measure the global properties of singular objects, and to measure the relative position of different objects, in terms of distance and direction.
Example –
length (of an arc) and distance (of a point to point).

2. Dynamic Spatial Operators :
Dynamic operations changes the objects upon which the operators are applied. Create, destroy, and update are the fundamental dynamic operations.
Example –
Updation of a spatial object via **translate**, **rotate**, **scale up or scale down**, **reflect**, and **shear**.

Table: Main Spatial Operators

Operator	Description
SDO_FILTER	Specifies which geometries may interact with a given geometry.
SDO_JOIN	Performs a spatial join based on one or more topological relationships.
SDO_NN	Determines the nearest neighbor geometries to a geometry.
SDO_NN_DISTANCE	Returns the distance of an object returned by the SDO_NN operator.
SDO_POINTINPOLYGON	Takes a set of rows whose first column is a point's x-coordinate value and the second column is a point's y-coordinate value, and returns those rows that are within a specified polygon geometry.
SDO_RELATE	Determines whether or not two geometries interact in a specified way.
SDO_WITHIN_DISTANCE	Determines if two geometries are within a specified distance from one another.

Table : lists operators, provided for convenience, that perform an SDO_RELATE operation of a specific mask type.

Spatial Queries

- It is a set of **spatial** conditions characterized by **spatial** operators that form the basis for the retrieval of **spatial** information from a **spatial** database system
- A request expressed as a combination of **spatial** conditions (e.g., Euclidean distance from a **query** point) for extracting specific information from a large amount of **spatial** data without actually changing these data
- A **spatial query** is a special type of database **query** supported by geodatabases. The queries differ from SQL queries in several important ways. Two of the most important are that they allow for the use of geometry data types such as points, lines and polygons and that these queries consider the **spatial** relationship between these geometries
- A **spatial query** uses properties and/or relationships that are of **spatial** nature and are not explicitly available in the BIM. To process a **spatial query** the 3D geometry model is analyzed

The requests for the spatial data which requires the use of spatial operations are called Spatial Queries.

It can be divided into –

1. **Range queries :**
   It finds all objects of a particular type that are within a given spatial area.
   Example –
   Finds all hospitals within the Siliguri area. A variation of this query is for a given location, find all objects within a particular distance, for example, find all banks within 5 km range.
2. **Nearest neighbor queries :**
   It Finds object of a particular type which is nearest to a given location.
   Example –
   Finds the nearest police station from the location of accident.
3. **Spatial joins or overlays :**
   It joins the objects of two types based on spatial condition, such as the objects which are intersecting or overlapping spatially.
   Example –
   Finds all Dhabas on a National Highway between two cities. It spatially joins township objects and highway object.
   Finds all hotels that are within 5 kilometres of a railway station. It spatially joins railway station objects and hotels objects.

2.6 Spatial Indexing

- A **spatial index** is a specialized **index**ing structure where the **index**ing key is the **spatial** location of objects **index**ed. The type of searches on an **spatial index** is the set of **spatial** queries, for example, range and overlapping queries.

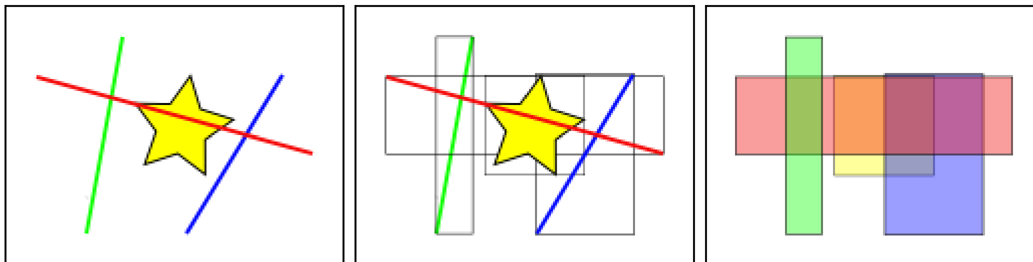
- Spatial indexing method divides the space into manageable number of smaller subspaces, which can be further divided into smaller subspaces and so on. The partitioning continues until the unpartitioned subspace contains the objects that can be stored in a data page. While designing the index structures for spatial databases the storage space must be efficiently utilized and the information retrieval should be fast and easy

- A **spatial index** is a specialized **index**ing structure where the **index**ing key is the **spatial** location of objects **index**ed. The type of searches on an **spatial index** is the set of **spatial** queries, for example, range and overlapping queries

```
CREATE INDEX nyc_census_blocks_geom_idx
  ON nyc_census_blocks
  USING GIST (geom);
```

How Spatial Indexes Work

Standard database indexes create a hierarchical tree based on the values of the column being indexed. Spatial indexes are a little different – they are unable to index the geometric features themselves and instead index the bounding boxes of the features.

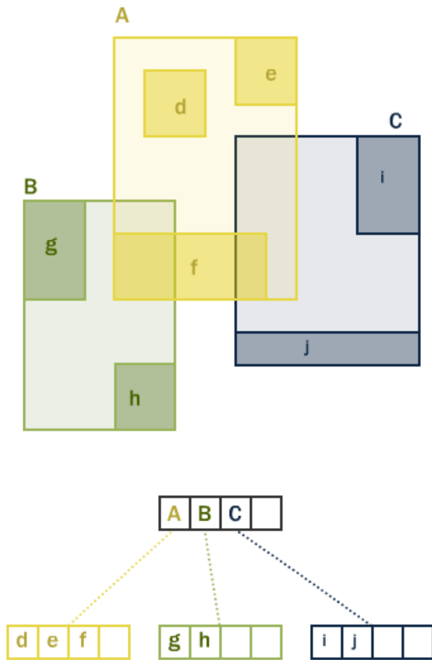


In the figure above, the number of lines that intersect the yellow star is **one**, the red line. But the bounding boxes of features that intersect the yellow box is **two**, the red and blue ones.

The way the database efficiently answers the question "what lines intersect the yellow star" is to first answer the question "what boxes intersect the yellow box" using the index (which is very fast) and then do an exact calculation of "what lines intersect the yellow star" **only for those features returned by the first test**.

For a large table, this "two pass" system of evaluating the approximate index first, then carrying out an exact test can radically reduce the amount of calculations necessary to answer a query.

Both PostGIS and Oracle Spatial share the same "R-Tree" 1 spatial index structure. R-Trees break up data into rectangles, and sub-rectangles, and sub-sub rectangles, etc. It is a self-tuning index structure that automatically handles variable data density, differing amounts of object overlap, and object size.

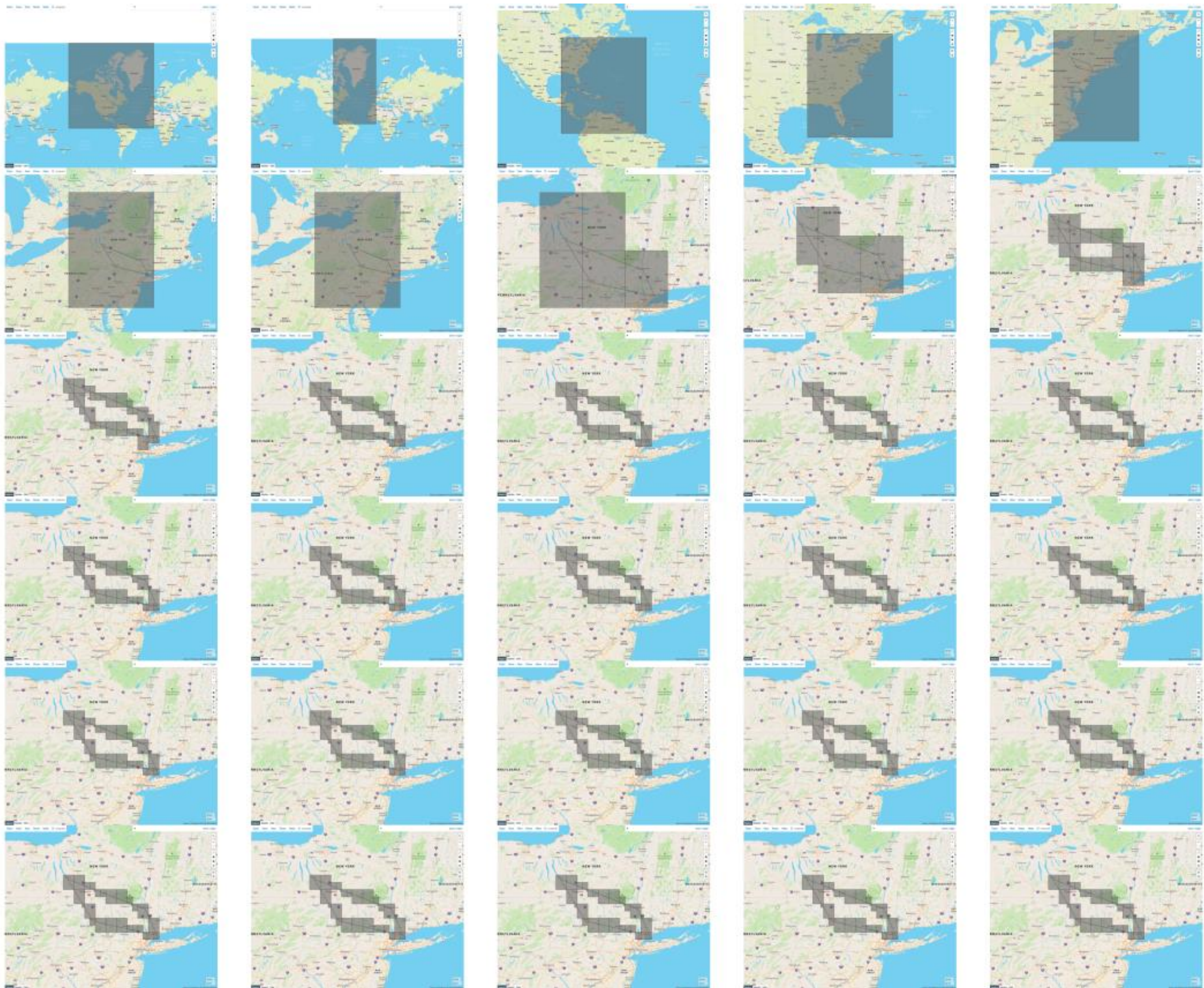


Data structures like B trees have been designed for efficient insertion and deletion in databases.

Spatial indexing is used to look up the values that match the predicate in efficient manner. There are two ways to provide spatial indexing:

i.) Dedicated external spatial data structures are added to the system that provide the attributes for spatial databases e.g. a B-tree does for standard attributes, and

ii.) spatial objects are mapped into a one-dimensional space so that they can be stored within a standard one dimensional index such as a B-tree.



SPATIAL DATA MINING

Spatial data mining refers to the process of the retrieval of information or patterns that are not explicitly stored in the spatial databases. Spatial data mining methods are used for the better understanding of spatial data, identifying the relationships between spatial data and non- spatial data, query optimization in spatial databases etc.

Statistical Spatial analysis is the most commonly and widely used data mining technique. It assumes that the spatial data are independent which in fact is not true as the spatial data are interrelated with their neighboring objects. Statistical method cannot handle symbolic values and non linear rules and are also very costly in the result computation. Several Machine learning techniques like learning from examples and generalization and specialization are used in spatial data mining.

SPATIAL DATA MINING ARCHITECTURE

Matheus architecture is the most general and widely used architecture in spatial data mining. This architecture is user controlled. All the predefined information about the objects is stored in the knowledge base which is fetched by the DB interface for query optimization. The information which

is useful for the pattern recognition is decided by the Focus Component and fed as input to the pattern extraction. The output is then monitored and evaluated by Evaluation module and duplicate values are removed. All the components interact using the Controller.

Geographic data consists of the spatial objects and the non spatial information about these objects (which can be stored in database as a pointer to the spatial description of object). Spatial data is characterised by geometric as well as topological characteristics where geometric characteristics involve the information about length, area, perimeter etc and topological characteristics include the information about neighbours, intrsection etc.

SPATIAL DATA MINING METHODS

Various methods have been designed for mining the data related to geometric space like points, polygon, rectangles, network and other complex objects. There are various kinds of rules associated with spatial data mining.

a.) Characteristic Rule: It refers to the general description of object data. Example rule describing the general price range of shops in various geographic regions of a city.

b.) Discriminant Rule: It refers to the properties or features that distinguish one object from other. Example the comparison of the various shops prices in different regions.

c.) Association Rule: It refers to the association of one object with other.

2.7 Applications

The following are examples of the kinds of data mining applications that could benefit from including spatial information in their processing:

- **Business prospecting**: Determine if colocation of a business with another franchise (such as colocation of a Pizza Hut restaurant with a Blockbuster video store) might improve its sales.
- **Store prospecting**: Find a good store location that is within 50 miles of a major city and inside a state with no sales tax. (Although 50 miles is probably too far to drive to avoid a sales tax, many customers may live near the edge of the 50-mile radius and thus be near the state with no sales tax.)
- **Hospital prospecting**: Identify the best locations for opening new hospitals based on the population of patients who live in each neighborhood.
- **Hotspot Detection**-Given a set of geospatial points which are related to an activity in a spatial domain, hotspots are the regions that are more active and have higher density of points compared to other regions.
- **Spatial region**-based classification or personalization: Determine if southeastern United States customers in a certain age or income category are more likely to prefer "soft" or "hard" rock music.
- **Automobile insurance**: Given a customer's home or work location, determine if it is in an area with high or low rates of accident claims or auto thefts.

- **Property analysis**: Use colocation rules to find hidden associations between proximity to a highway and either the price of a house or the sales volume of a store.
- **Property assessment**: In assessing the value of a house, examine the values of similar houses in a neighborhood, and derive an estimate based on variations and spatial correlation.

2.8 Mobile Databases: Location and Handoff Management

Location Management: In cellular systems a mobile unit is free to move around within the entire area of coverage. Its movement is random and therefore its geographical location is unpredictable. This situation makes it necessary to locate the mobile unit and record its location to HLR and VLR when a call has to be delivered to it.

Thus, the entire process of the mobility management component of the cellular system is responsible for two tasks:

(a) **location management**- identification of the current geographical location or current point of attachment of a mobile unit which is required by the MSC (Mobile Switching Center) to route the call.

(b) **handoff-** transferring (handing off) the current (active) communication session to the next base station, which seamlessly resumes the session using its ownset of channels.

One of the main objectives of efficient location management schemes is to **minimize the communication overhead due to database updates (mainly HLR).**

The current point of location of a subscriber (mobile unit) is expressed in terms of the cell or the base station to which it is presently connected. The mobile units (called and calling subscribers) can continue to talk and move around in their respective cells; but as soon as both or any one of the units moves to a different cell,the location management procedure is invoked to identify the new location.

The location management performs three fundamental tasks:

(a) location update, (b) location lookup, and (c) paging.

Location update-is initiated by the mobile unit, the current location of the unit is recorded in HLR and VLR databases.

Location lookup- a database search to obtain the current location of the mobile unit.

Paging -the system informs the caller **the location of the called unit** in termsof its current base station. These two tasks are initiated by the MSC .

The cost of update and paging increases as cell size decreases,which becomes quite significant for finer granularity cells such as micro- or picocell clusters. The presence of **frequent cell crossing,** which is a common scenario in highly commuting zones, further adds to the cost. The system creates location areas and paging areas to **minimize the cost.**

A number of neighbouring cells are grouped together to form a **location area**, and the paging area is constructed in a similar way. It is useful to keep the same set of cells for creating location and paging areas, and in most commercial systems they are usually identical. This arrangement reduces location update frequency because location updates are not necessary when a mobile unit moves in the cells of a location area. A large number of schemes to achieve low cost and infrequent update have been proposed, and new schemes continue to emerge as cellular technology advances.

A mobile unit can freely move around in

(a) active mode, (b) doze mode, or (c) power down mode.

In **active mode,** the mobile actively communicates with other subscriber, and it may continue to move within the cell or may encounter a handoff which may interrupt the communication. It is the task of the location manager to find the new location and resume the communication.

In **doze mode** a mobile unit does not actively communicate with other subscribers but continues to listen to the base station and monitors the signal levels around it

In **Power down mode** the unit is not functional at all.

When it moves to a different cell in doze or power down modes,then it is neither possible nor necessary for the location manager to find the location.

The **location management module** uses a **two-tier scheme** for location- related tasks. The first tier provides a quick location lookup, and the second tier search is initiated only when the first tier search fails.

Location Lookup:A location lookup finds the location of **the called party** to establish the communication session. It involves searching VLR and possibly HLR. Figure 3.1 illustrates the

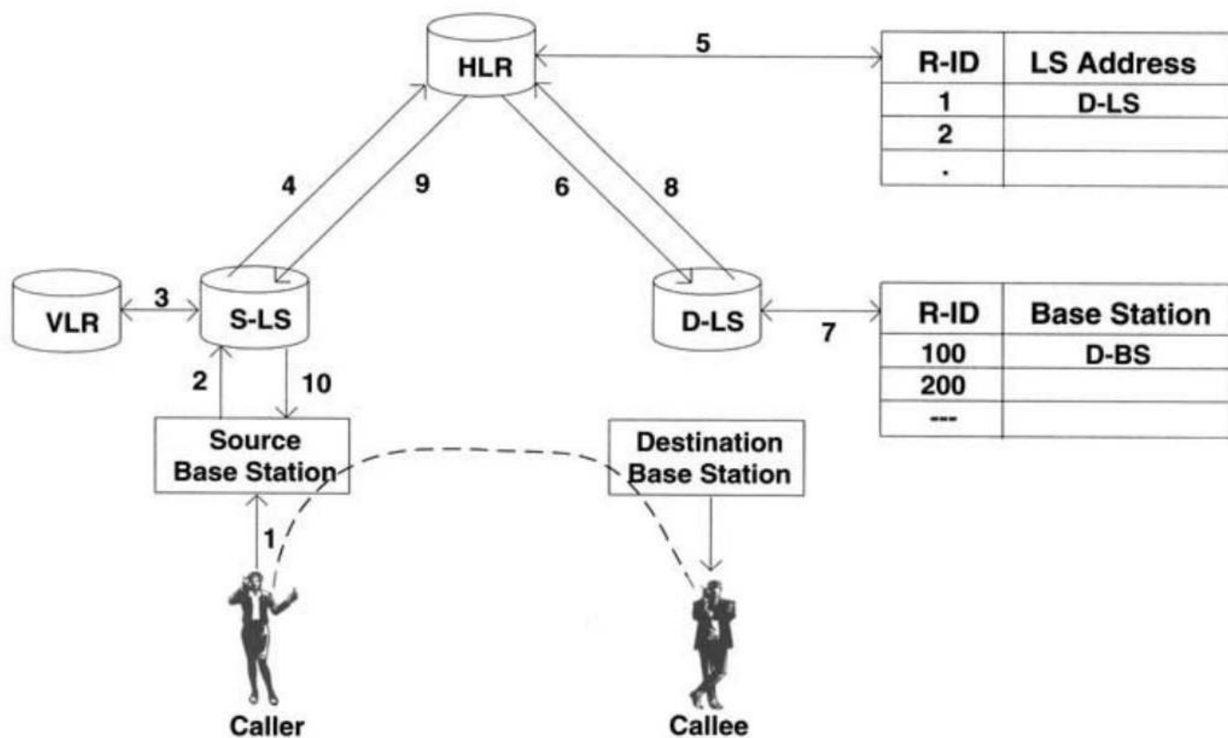


Fig. 3.1 Location search steps.

Step 1: The caller dials a number. To find the location of the called number (destination), the caller unit sends a location query to its base station **source base station**.

Step 2: The source base station sends the query to the S-LS (source location server) for location discovery.

Step 3: S-LS first looks up the VLR to find the location. If the called number is a visitor to the source base station, then the location is known and the connection is set up.

Step 4: If VLR search fails, then the location query is sent to the HLR.

Step 5: HLR finds the location of D-LS (destination location server).

Step 6: The search goes to D-LS.

Step 7: D-LS finds the address of D-BS (destination base station).

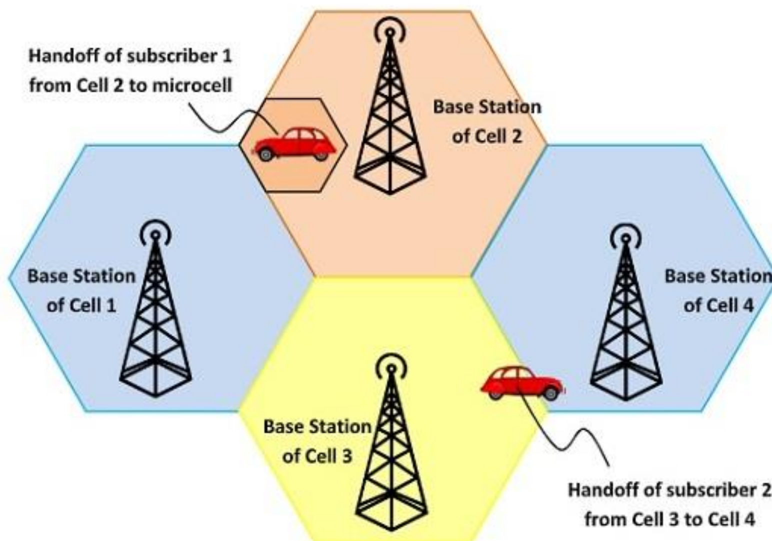
Step 8: Address of D-BS is sent to he HLR.

Step 9: HLR sends the address of D-BS to S-LS (source location server).

Step 10: The address of D-BS is sent to the source base station, which sets up the communication session.

Handoff technology

In cellular communications, the handoff is the process of transferring an active call or data session from one cell in a cellular network or from one channel to another. In satellite communications, it is the process of transferring control from one earth station to another. Handoff is necessary for preventing loss of interruption of service to a caller or a data session user. Handoff is also called handover.



Situations for triggering Handoff

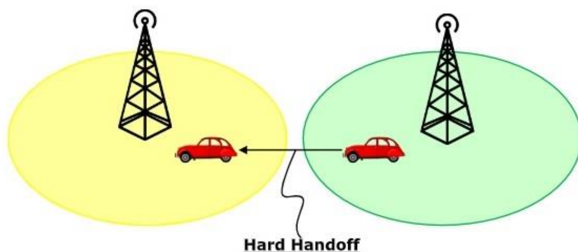
Handoffs are triggered in any of the following situations −

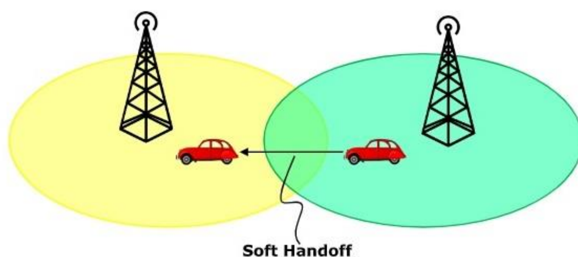
- If a subscriber who is in a call or a data session moves out of coverage of one cell and enters coverage area of another cell, a handoff is triggered for a continuum of service. The tasks that were being performed by the first cell are delineating to the latter cell.
- Each cell has a pre-defined capacity, i.e. it can handle only a specific number of subscribers. If the number of users using a particular cell reaches its maximum capacity, then a handoff occurs. Some of the calls are transferred to adjoining cells, provided that the subscriber is in the overlapping coverage area of both the cells.
- Cells are often sub-divided into microcells. A handoff may occur when there is a transfer of duties from the large cell to the smaller cell and vice versa. For example, there is a traveling user moving within the jurisdiction of a large cell. If the traveler stops, then the jurisdiction is transferred to a microcell to relieve the load on the large cell.
- Handoffs may also occur when there is an interference of calls using the same frequency for communication.

Types of Handoffs

There are two types of handoffs −

- **Hard Handoff** − In a hard handoff, an actual break in the connection occurs while switching from one cell to another. The radio links from the mobile station to the existing cell is broken before establishing a link with the next cell. It is generally an inter-frequency handoff. It is a "break before make" policy.
- **Soft Handoff** − In soft handoff, at least one of the links is kept when radio links are added and removed to the mobile station. This ensures that during the handoff, no break occurs. This is generally adopted in co-located sites. It is a "make before break" policy.





- Mobile Assisted Handoff

Mobile Assisted Handoff (MAHO) is a technique in which the mobile devices assist the Base Station Controller (BSC) to transfer a call to another BSC. It is used in GSM cellular networks. In other

systems, like AMPS, a handoff is solely the job of the BSC and the Mobile Switching Centre (MSC), without any participation of the mobile device. However, in GSM, when a mobile station is not using its time slots for communicating, it measures signal quality to nearby BSC and sends this information to the BSC. The BSC performs handoff according to this information.

2.9 Deductive database

A **deductive database** is a database system that can make deductions (i.e. conclude additional facts) based on rules and facts stored in the (deductive) database. Datalog is the language typically used to specify facts, rules and queries in deductive databases. Deductive databases have grown out of the desire to combine logic programming with relational databases to construct systems that support a powerful formalism and are still fast and able to deal with very large datasets. Deductive databases are more expressive than relational databases but less expressive than logic programming systems. In recent years, deductive databases such as Datalog have found new application in data integration, information extraction, networking, program analysis, security, and cloud computing.

Deductive databases reuse many concepts from logic programming; rules and facts specified in the deductive database language Datalog look very similar to those in Prolog. However important differences between deductive databases and logic programming:

- Order sensitivity and procedurality: In Prolog, program execution depends on the order of rules in the program and on the order of parts of rules; these properties are used by programmers to build efficient programs. In database languages (like SQL or Datalog), however, program execution is independent of the order of rules and facts.
- Special predicates: In Prolog, programmers can directly influence the procedural evaluation of the program with special predicates such as the cut, this has no correspondence in deductive databases.
- Function symbols: Logic Programming languages allow function symbols to build up complex symbols. This is not allowed in deductive databases.
- Tuple-oriented processing: Deductive databases use set-oriented processing while logic programming languages concentrate on one tuple at a time.

A **Deductive Database** is a type of database that can make conclusions or we can say deductions using a sets of well defined rules and fact that are stored in the database. In today's world as we deal with a large amount of data, this deductive database provides a lot of advantages. It helps to combine the *RDBMS* with logic programming. To design a deductive database a purely declarative programming language called Datalog is used.

The implementations of deductive databases can be seen in LDL (Logic Data Language), NAIL (Not Another Implementation of Logic), CORAL, and VALIDITY.
The use of LDL and VALIDITY in a variety of business/industrial applications

1. LDL Applications:
This system has been applied to the following application domains:

- Enterprise modelling
- Hypothesis testing or data dredging
- Software reuse

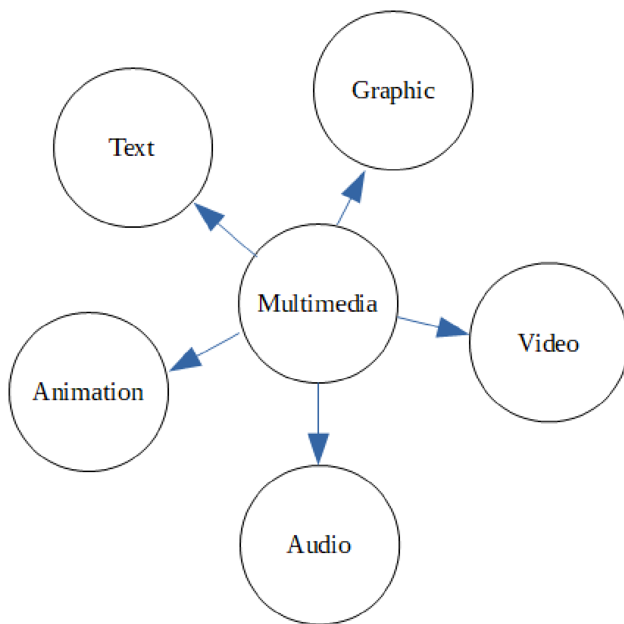
2. VALIDITY Applications:

- Electronic commerc
- Rules-governed processes

- Knowledge discovery
- Concurrent Engineering

2.10 Multimedia Database

The multimedia databases are used to store multimedia data such as images, animation, audio, video along with text. This data is stored in the form of multiple file types like .txt(text), .jpg(images), .swf(videos), .mp3(audio) etc.



Contents of the Multimedia Database

The multimedia database stored the multimedia data and information related to it. This is given in detail as follows −

Media data

This is the multimedia data that is stored in the database such as images, videos, audios, animation etc.

Media format data

The Media format data contains the formatting information related to the media data such as sampling rate, frame rate, encoding scheme etc.

Media keyword data

This contains the keyword data related to the media in the database. For an image the keyword data can be date and time of the image, description of the image etc.

Media feature data

Th Media feature data describes the features of the media data. For an image, feature data can be colours of the image, textures in the image etc.

Challenges of Multimedia Database

There are many challenges to implement a multimedia database. Some of these are:

- Multimedia databases contains data in a large type of formats such as .txt(text), .jpg(images), .swf(videos), .mp3(audio) etc. It is difficult to convert one type of data format to another.
- The multimedia database requires a large size as the multimedia data is quite large and needs to be stored successfully in the database.
- It takes a lot of time to process multimedia data so multimedia database is slow.

Multimedia databases are the main source of interaction between users and multimedia elements.

Multimedia storage is characterised by the following −

- Massive storage volumes.
- Large object sizes.
- Multiple related objects.
- Temporal requirements for retrieval.

A multimedia database system stores and manages a large collection of multimedia data, such as audio, video, image, graphics, speech, text, document, and hypertext data, which contain text, text markups, and linkages. Multimedia database systems are increasingly common owing to the popular use of audio-video equipment, digital cameras, CD-ROMs, and the Internet. There are multimedia database systems include NASA's EOS (Earth Observation System), various kinds of image and audio video databases, and Internet databases.

There is two main groups of multimedia indexing and retrieval systems which are as follows –

Description-based retrieval systems − It is used to build indices and perform object retrieval based on image descriptions, such as keywords, captions, size, and time of creation. Description-based retrieval is labor-intensive if performed manually. If automated, the results are typical of poor quality.

For instance, the assignment of keywords to images can be a difficult and arbitrary service. The latest development of Web-based image clustering and classification techniques has enhanced the quality of definition-based Web image retrieval because image surrounded text information and Web linkage information can be used to extract proper description and group images describing a similar theme together.

Content-based retrieval systems − It can support retrieval based on the image content, such as color histogram, texture, pattern, image topology, and the shape of objects and their layouts and locations within the image. Content-based retrieval facilitates visual characteristics to index images and improves object retrieval based on feature similarity, which is highly desirable in several applications.

In a content-based image retrieval system, there are often two kinds of queries − image sample-based queries and image feature specification queries. Image-sample-based queries find all of the images that are similar to the given image sample. This search analyzes the feature vector (or signature) extracted from the sample with the feature vectors of images that have been extracted and ordered in the image database.

UNIT III NOSQL DATABASES

NoSQL – CAP Theorem – Sharding - Document based – MongoDB Operation: Insert, Update, Delete, Query, Indexing, Application, Replication, Sharding–Cassandra: Data Model, Key Space, Table Operations, CRUD Operations, CQL Types – HIVE: Data types, Database Operations, Partitioning – HiveQL – OrientDB Graph database – OrientDB Features

NOSQL DATABASES

NoSQL databases use a variety of data models for accessing and managing data. These types of databases are optimized specifically for applications that require large data volume, low latency, and flexible data models, which are achieved by relaxing some of the data consistency restrictions of other databases.

3.1 NOSQL

Understanding Differences in the Four Types of NoSQL Databases

- Document databases.
- Key-value stores.
- Column-oriented databases.
- Graph databases.

3.2 CAP THEOREM - SHARDING - DOCUMENT BASED

CAP Theorem is a concept that a distributed database system can only have 2 of the 3: Consistency, Availability and Partition Tolerance.

Understanding CAP theorem with an Example

CAP theorem, also known as **Brewer's theorem**, stands for **Consistency, Availability and Partition Tolerance.** But let's try to understand each, with an example.

Availability

because of the amazing plans it offers. Besides that, they also provide an amazing customer care service where you can call anytime and get your queries and concerns answered quickly and efficiently.

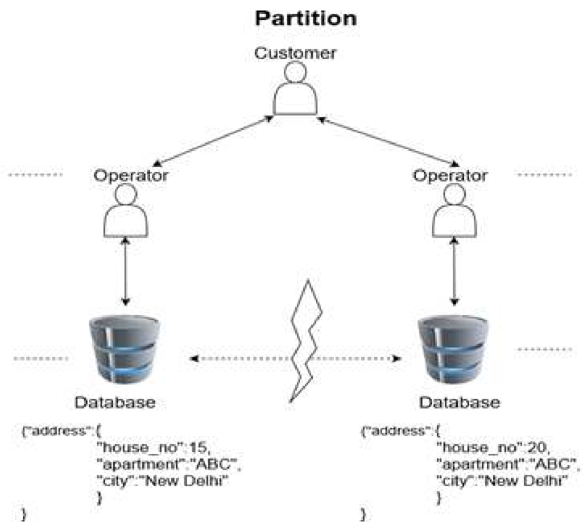
- Whenever a customer calls them, the mobile operator is able to connect them to one of their customer care operators.

- The customer is able to elicit any information required by her/him about his accounts like balance, usage, or other information. We call this **Availability** because every customer is able to connect to the operator and get the information about the user/customer.

- For example, when you visit your bank's ATM, you are able to access your account and its related information. Now even if you go to some other ATM, you should still be able to access your account.

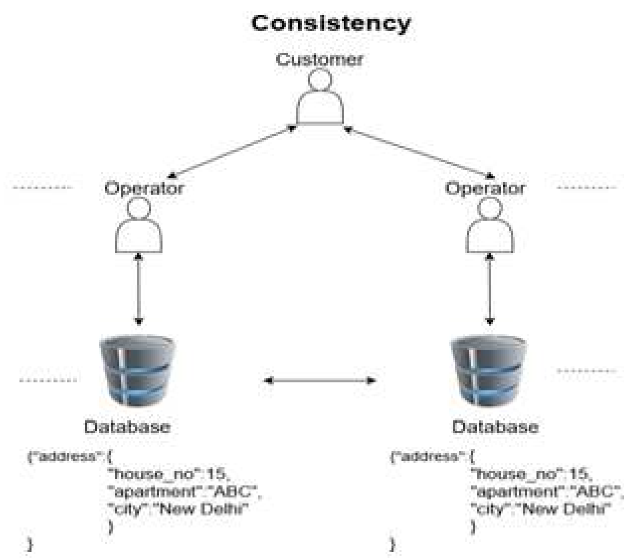
- For example, you should be able to see your friend's Whatsapp status even if you are viewing an outdated one due to some network failure.

- **Partition tolerance** means that the system can continue operating if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other.
- Recently you have noticed that your current mobile plan does not suit you. You do not access that much mobile data any longer because you have good wi-fi facilities at home and at the office, and you hardly step outside anywhere. Therefore, you want to update your mobile plan. So you decide to call the customer care once again.



- On connecting with the operator this time, they tell you that they have not been able to update their records due to some issues. So the information lying with the operator might not be up to date, therefore they cannot update the information. We can say here that the service is broken or there is no **Partition tolerance.**

- **Consistency** means that the nodes will have the same copies of a replicated data item visible for various transactions.



Now, you have recently shifted to a new house in the city and you want to update your address registered with the mobile operator. You decide to call the customer care operator and update it with them. When you call, you connect with an operator. This operator makes the relevant changes in the system. But once you have put down the phone, you realize you told them the correct street name but the old house number.

So you frantically call the customer care again. This time when you call, you connect with a different customer care operator but they are able to access your records as well and know that you have recently updated your address. They make the relevant changes in the house

operator, they were able to retrieve the same information.

For example, your bank account should reflect the same balance whether you view it from your PC, tablet, or smartphone!

What is the CAP Theorem?

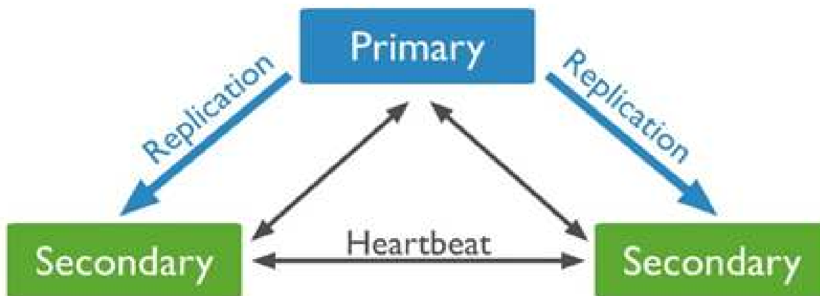
In the last section, you understood what each term means in the CAP theorem. Now let us understand the theorem itself.

"The CAP theorem states that a distributed database system has to make a

tradeoff between Consistency and Availability when a Partition occurs." A distributed database system is bound to have partitions in a real-world system due to network failure or some other reason. Therefore, partition tolerance is a property we cannot avoid while building our system. So a distributed system will either choose to give up on Consistency or Availability but not on Partition tolerance.

Understanding CP with MongoDB

Let's try to understand how a distributed system would work when it decides to give up on Availability during a partition with the help of MongoDB.



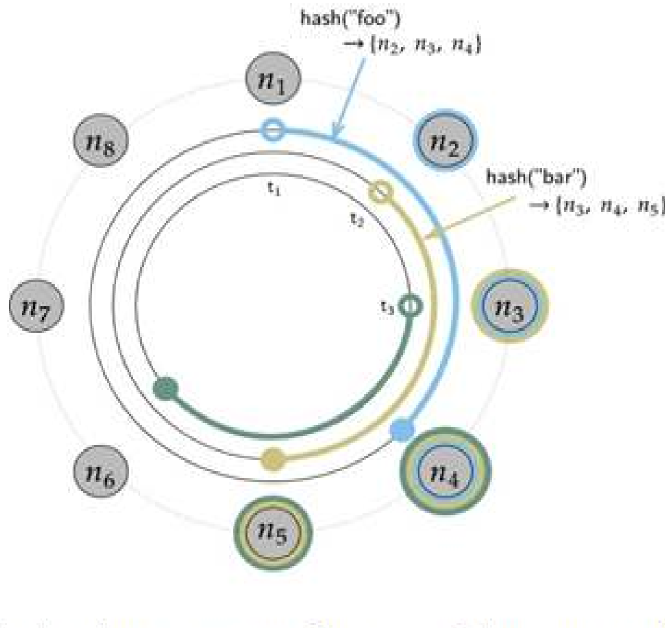
MongoDB is a NoSQL database that stores data in one or more Primary nodes in the form of JSON files. Each Primary node has multiple replica sets that update themselves asynchronously using the operation log file of their respective primary node. The replica set nodes in the system send a heartbeat (ping) to every other node to keep track if other replicas or primary nodes are alive or dead. If no heartbeat is received within 10 seconds, then that node is marked as inaccessible.

If a Primary node becomes inaccessible, then one of the secondary nodes needs to become the primary node. Till a new primary is elected from amongst the secondary nodes, the system remains unavailable to the user to make any new write query. Therefore, the MongoDB system behaves as a Consistent system and compromises on Availability during a partition.

Understanding AP with Cassandra

Cassandra is a peer-to-peer system. It consists of multiple nodes in the system. And each node can accept a read or write request from the user. Cassandra maintains multiple replicas of data in separate nodes. This gives it a masterless node architecture where there are multiple points of failure instead of a single point.

The replication factor determines the number of replicas of data. If the replication factor is 3, then we will replicate the data in three nodes in a clockwise manner.



A situation can occur where a partition occurs and the replica does not get an updated copy of the data. In such a situation the replica nodes will still be available to the user but the data will be inconsistent. However, Cassandra also provides eventual consistency. Meaning, all updates will reach all the replicas eventually. But in the meantime, it allows divergent versions of the same data to exist temporarily. Until we update them to the consistent state.

Therefore, by allowing nodes to be available throughout and allowing temporarily inconsistent data to existing in the system, Cassandra is an AP database that compromises on consistency.

3.3 MONGODB OPERATION: INSERT, UPDATE, DELETE, QUERY, INDEXING, APPLICATION, REPLICATION, SHARDING

MongoDB

MongoDB is a No SQL database. It is an open-source, cross-platform, document-oriented database written in C++.

What is MongoDB

- MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling.
- Mongo DB is a document-oriented database. It is an open source product, developed and supported by a company named 10gen.
- MongoDB is available under General Public license for free, and it is also

MongoDB was designed to work with commodity servers. Now it is used by the company of all sizes, across all industry.
MongoDB CRUD Operations

- Create Operations
- Read Operations
- Update Operations
- Delete Operations
- Bulk Write

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.
For examples, see Insert Documents.

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.
For examples, see:

- Query Documents
- Query on Embedded/Nested Documents
- Query an Array
- Query an Array of Embedded Documents

Update Operations

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

- db.collection.updateOne()
- db.collection.updateMany()
- db.collection.replaceOne()

- 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.
For examples, see Update Documents.

- db.collection.deleteOne()
- db.collection.deleteMany()
- 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.
For examples, see Delete Documents.
Purpose of building MongoDB

It may be a very genuine question that - "what was the need of MongoDB although there were many databases in action?"
There is a simple answer:

All the modern applications require big data, fast features development, flexible deployment, and the older database systems not competent enough, so the MongoDB was needed.
The primary purpose of building MongoDB is:

- Scalability
- Performance
- High Availability
- Scaling from single server deployments to large, complex multi-site architectures.
- Key points of MongoDB
- Develop Faster
- Deploy Easier
- Scale Bigger

Example of document oriented database

- MongoDB is a document oriented database. It is a key feature of MongoDB. It offers a document oriented storage. It is very simple you can program it easily.
- MongoDB stores data as documents, so it is known as document-oriented database.

1. FirstName = "John",
2. Address = "Detroit",
3. Spouse = [{Name: "Angela"}].
4. FirstName ="John",
5. Address = "Wick"

There are two different documents (separated by ".").

Storing data in this manner is called as document-oriented database.
Mongo DB falls into a class of databases that calls Document Oriented Databases. There is also a broad category of database known as **No SQL Databases**.

MongoDB insert documents

In MongoDB, the **db.collection.insert()** method is used to add or insert new documents into a collection in your database.
Insert

Insert is an operation that performs either an update of existing document or an insert of new document if the document to modify does not exist.

Syntax

```
1.>db.COLLECTION_NAME.insert(document)
```

Lets take an example to demonstrate how to insert a document into a collection. In this example we insert a document into a collection named javatpoint. This operation will automatically create a collection if the collection does not currently exist.

Example

```
1.   db.javatpoint.inse
rt( 2. {
3.        course: "java",
4.        details: {
5.           duration: "6 months",
6.           Trainer: "Sonoo jaiswal"
7.        },
8.        Batch: [ { size: "Small", qty: 15 }, { size: "Medium", qty: 25 } ],
9.        category: "Programming language"
10.    }
11.)
```

After the successful insertion of the document, the operation will return a WriteResult object with its status.

Output:

```
WriteResult({ "nInserted" : 1 })
```

documents inserted. If an error is occurred then the **WriteResult** will specify the error information.

Check the inserted documents

If the insertion is successful, you can view the inserted document by the following query.

```
1.   >db.javatpoint.find()
```

You will get the inserted document in return.

Output:

```
{ "_id" : ObjectId("56482d3e27e53d2dbc93cef8"), "course" : "java", "details" :
{ "duration" : "6 months", "Trainer" : "Sonoo jaiswal" }, "Batch" :
```

Note: Here, the ObjectId value is generated by MongoDB itself. It may differ from the one shown.

MongoDB insert multiple documents

If you want to insert multiple documents in a collection, you have to pass an array of documents

Define a variable named Allcourses that hold an array of documents to insert.

```
var
    Allcours
    es = [

      {
         Course: "Java",
               details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
         Batch: [ { size: "Medium", qty: 25 } ],
         category: "Programming Language"
       },
       {
         Course: ".Net",
         details: { Duration: "6 months", Trainer: "Prashant Verma" },
         Batch: [ { size: "Small", qty: 5 }, { size: "Medium", qty: 10 }, ],
         category: "Programming Language"
       },
       {
         Course: "Web Designing",
         details: { Duration: "3 months", Trainer: "Rashmi Desai" },
         Batch: [ { size: "Small", qty: 5 }, { size: "Large", qty: 10 } ],
         category: "Programming Language"
  }
 ];
```

Inserts the documents

Pass this Allcourses array to the db.collection.insert() method to perform a bulk insert.

1. > db.javatpoint.**insert**(Allcourses);

After the successful insertion of the documents, this will return a BulkWriteResult object

```
BulkWriteResult({ "writeErrors
   " : [ ],
   "writeConcernErrors" : [ ],
   "nInserted" : 3,

   "nUpserted" : 0,

   "nMatched" : 0,

})
```

with the status.

Note: Here the nInserted field specifies the number of documents inserted. In the case of any error during the operation, the **BulkWriteResult** will specify that error.

You can check the inserted documents by using the following query:

1. >db.javatpoint.find()

MongoDB Advantages

- **MongoDB is schema less**. It is a document database in which one collection holds different documents

- There are no complex joins in MongoDB.
- MongoDB provides the **facility of deep query** because it supports a powerful dynamic query on documents.
- It is very **easy to scale**.
- It **uses internal memory for storing working sets** and this is the reason of its fast access.

Distinctive features of MongoDB

These are some important features of MongoDB:

1. Support ad hoc queries: In MongoDB, you can search by field, range query and it also supports regular expression searches.
2. Indexing: You can index any field in a document.
3. Replication: Mongo DB supports Master Slave replication. A master can perform Reads and Writes and a Slave copies data from the master and can only be used for reads or back up (not writes)
4. Duplication of data : Mongo DB can run over multiple servers. The data is duplicated to keep the system up and also keep its running condition in case of hardware failure.
5. Load balancing: It has an automatic load balancing configuration because of data placed in shards.
6. Supports map reduce and aggregation tools.
7. Uses JavaScript instead of Procedures.
8. It is a schema-less database written in C++.
9. Provides high performance.
10. Stores files of any size easily without complicating your stack.
11. Easy to administer in the case of failures.
12. It also supports:
    - JSON data model with dynamic schemas
    - Auto-sharding for horizontal scalability
    - Built in replication for high availability

Now a day many companies using MongoDB to create new types of applications improve performance and availability.

Where MongoDB should be used

- Big and complex data
- Mobile and social infrastructure
- Content management and delivery
- User data management
- Data hub

MongoDB Datatypes

Following is a list of usable data types in MongoDB.

Data Types	Description
String	String is the most commonly used datatype. It is used to store data. A string must be UTF 8 valid in mongodb.

Boolean	This datatype is used to store boolean values. It just shows YES/NO values.
Double	Double datatype stores floating point values.
Min/Max Keys	This datatype compare a value against the lowest and highest bson elements.
Arrays	This datatype is used to store a list or multiple values into a single key.
Object	Object datatype is used for embedded documents.
Null	It is used to store null values.
Symbol	It is generally used for languages that use a specific type.
Date	This datatype stores the current date or time in unix time format. It makes you possible to specify your own date time by creating object of date and pass the value of date, month, year into it.

MongoDB Create Database

Use Database method:

- ✓ There is no create database command in MongoDB. Actually, MongoDB do not provide any command to create database.
- ✓ It may be look like a weird concept, if you are from traditional SQL background where you need to create a database, table and insert values in the table manually.
- ✓ Here one thing is very remarkable that you can create collection manually by "db.createCollection()" but not the database.
- ✓ Here, in MongoDB you don't need to create a database manually because MongoDB will create it automatically when you save the value into the defined collection at first time.

How and when to create database

If there is no existing database, the following command is used to create a new database.

Syntax: use DATABASE_NAME

If the database already exists, it will return the existing database.

Let' take an example to demonstrate how a database is created in MongoDB. In the following example, we are going to create a database "javatpointdb".

See this example

```
>use javatpointdb
Swithched to db javatpointdb
```

To **check the currently selected database**, use the command db:

```
>db
javatpointdb
```

document into it to display database:

```
>db.movie.insert({"name":"javatpoint"})
WriteResult({ "nInserted": 1})
```

```
>show dbs
```

```
javatpointdb 0.078GB
local 0.078GB
```

MongoDB Drop Database

The dropDatabase command is used to drop a database. It also deletes the associated data files. It operates on the current database.

Syntax: db.dropDatabase()

This syntax will delete the selected database. In the case you have not selected any database, it will delete

default "test" database.
To **check the database list**, use the command show dbs:

```
>show dbs
```

```
javatpointdb
0.078GB local
0.078GB
```

If you want to **delete the database "javatpointdb"**, use the dropDatabase() command as follow

```
>use javatpointdb
switched to the db javatpointdb
>db.dropDatabase()
{ "dropped": "javatpointdb", "ok": 1}
```

Now check the list of databases:

```
>show dbs
local 0.078GB
```

Cassandra vs MongoDB

Cassandra and MongoDB both are types of NoSQL databases. Cassandra is a distributed database system designed to handle large amount of data and known for its high scalability and high performance. While, MongoDB is document oriented database which also provides high scalability, high performance and automatic scaling.
In terms of simplicity, databases can be divided in two types:

- Development simplicity
- Operational simplicity

While MongoDB is known for an easy out-of-the-box experience, Cassandra is known for easy to manage at scale.
Following is a list of important differences between them:

1)	Cassandra is high performance distributed database system.	MongoDB is cross-platform document-oriented database system.
2)	Cassandra is written in Java.	MongoDB is written in C++.
3)	Cassandra stores data in tabular form like SQL format.	MongoDB stores data in JSON format.
4)	Cassandra is got license by Apache.	MongoDB is got license by AGPL and drivers by Apache.
5)	Cassandra is mainly designed to handle large amounts of data across many commodity servers.	MongoDB is designed to deal with JSON-like documents and access applications easier and faster.
6)	Cassandra provides high availability with no single point of failure.	MongoDB is easy to administer in the case of failure.

Php MongoDB Connectivity

Php provides **mongodb** driver to connect with mongoDB database. After installing it, we can perform database operations by using the php. Here, we are using Ubuntu 16.04 to create an example. This example includes the following steps.

1) Installing Driver

$ pecl install mongodb

2) Edit php.ini File

It is store in the apache server directory **/etc/php/7.0/apache2/php.ini**

$ extension = mongodb.so

```
Open ▾    php.ini
/etc/php/7.0/apache2                                    Save
;
;   extension=msql.so
;
; ... or with a path:
;
;   extension=/path/to/extension/msql.so
;
; If you only provide the name of the extension, PHP will look for it in its
; default extension directory.
;
; Windows Extensions
; Note that ODBC support is built in, so no dll is needed for it.
; Note that many DLL files are located in the extensions/ (PHP 4) ext/ (PHP 5+)
; extension folders as well as the separate PECL DLL download (PHP 5+).
; Be sure to appropriately set the extension_dir directive.
;
extension=mongodb.so
;extension=php_bz2.dll
;extension=php_curl.dll
;extension=php_fileinfo.dll
```

3) Install mongo-php library

$ composer require mongod

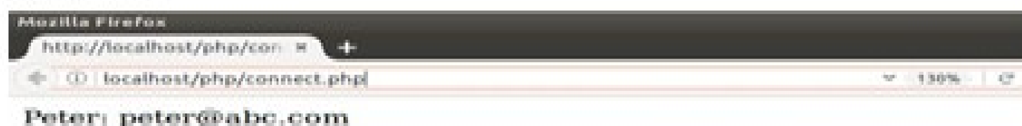
```
irfan@irfan-GB-BXBT-2807: ~
irfan@irfan-GB-BXBT-2807:~$ composer require mongodb/mongodb
PHP Warning:  Module 'mongodb' already loaded in Unknown on line 0
PHP Warning:  Module 'mongodb' already loaded in Unknown on line 0
PHP Warning:  Module 'mongodb' already loaded in Unknown on line 0
PHP Warning:  Module 'mongodb' already loaded in Unknown on line 0
Using version ^1.1 for mongodb/mongodb
./composer.json has been created
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing mongodb/mongodb (1.1.2) Downloading: 100%
Writing lock file
Generating autoload files
irfan@irfan-GB-BXBT-2807:~$
```

1) Create Php Script

// **connect.php**

```
1.  <?php
2.  require 'vendor/autoload.php';
3.  // Creating Connection
4.  $con = new MongoDB\Client("mongodb://localhost:27017");
5.  // Creating Database
6.  $db = $con->javatpoint;
7.  // Creating Document
$collection = $db->employee

8.  // Insering Record
9.  $collection->insertOne( [ 'name' =>'Peter', 'email' =>'peter@abc.com' ] );
10. // Fetching Record
11. $record = $collection->find( [ 'name' =>'Peter'] );
12. foreach ($record as $employe) {
13. echo $employe['name'], ': ', $employe['email']."<br>";
15.  }

16.  ?>
```

2Execute Php Script
Execute this script on the localhost server. It will create database and store data into the mongodb. localhost/php/**connect**.ph



$ mongo

```
root@irfan-GB-BXBT-2807: /home
root@irfan-GB-BXBT-2807:/home# mongo
MongoDB shell version: 3.2.13
connecting to: test
Server has startup warnings:
2017-05-20T14:46:45.854+0530 I CONTROL  [initandlisten]
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/
mm/transparent_hugepage/enabled is 'always'.
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] **        We suggest set
ting it to 'never'
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten]
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/
mm/transparent_hugepage/defrag is 'always'.
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] **        We suggest set
ting it to 'never'
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten]
>
```

6.1. Show Database

The following command is used to show databases.

> show dbs

```
root@irfan-GB-BXBT-2807: /home
root@irfan-GB-BXBT-2807:/home# mongo
MongoDB shell version: 3.2.13
connecting to: test
Server has startup warnings:
2017-05-20T14:46:45.854+0530 I CONTROL  [initandlisten]
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/
mm/transparent_hugepage/enabled is 'always'.
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] **        We suggest set
ting it to 'never'
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten]
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/
mm/transparent_hugepage/defrag is 'always'.
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] **        We suggest set
ting it to 'never'
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten]
> show dbs
javatpoint   0.000GB
local        0.000GB
>
```

6.2. Show Collection

The following command is used to show collections.

> show collections

```
root@irfan-GB-BXBT-2807: /home
root@irfan-GB-BXBT-2807:/home# mongo
MongoDB shell version: 3.2.13
connecting to: test
Server has startup warnings:
2017-05-20T14:46:45.854+0530 I CONTROL  [initandlisten]
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/
mm/transparent_hugepage/enabled is 'always'.
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] **        We suggest set
ting it to 'never'
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten]
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/
mm/transparent_hugepage/defrag is 'always'.
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] **        We suggest set
ting it to 'never'
2017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten]
> show dbs;
javatpoint  0.000GB
local       0.000GB
> use javatpoint
switched to db javatpoint
> show collections
employee
>
```

6.3 Access Records

> db.employee.find()

```
root@irfan-GB-BXBT-2807: /home
onnecting to: test
erver has startup warnings:
017-05-20T14:46:45.854+0530 I CONTROL  [initandlisten]
017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/
m/transparent_hugepage/enabled is 'always'.
017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] **        We suggest set
ing it to 'never'
017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten]
017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] ** WARNING: /sys/kernel/
m/transparent_hugepage/defrag is 'always'.
017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten] **        We suggest set
ing it to 'never'
017-05-20T14:46:45.855+0530 I CONTROL  [initandlisten]
 show dbs;
avatpoint  0.000GB
ocal       0.000GB
 use javatpoint
witched to db javatpoint
 show collections
mployee
 db.employee.find()
 "_id" : ObjectId("592015a93c10b904a57f05e2"), "name" : "Peter", "email" : "pet
r@abc.com" }
```

3.4 CASSANDRA: DATA MODEL, KEY SPACE, TABLE OPERATIONS, CRUD OPERATIONS, CQL TYPES

i. DATA MODEL

The data model of Cassandra is significantly different from what we normally see in an RDBMS. This chapter provides an overview of how Cassandra stores its data.

Cluster

Cassandra database is distributed over several machines that operate together. The outermost container is known as the Cluster. For failure handling, every node contains a replica, and in case of a

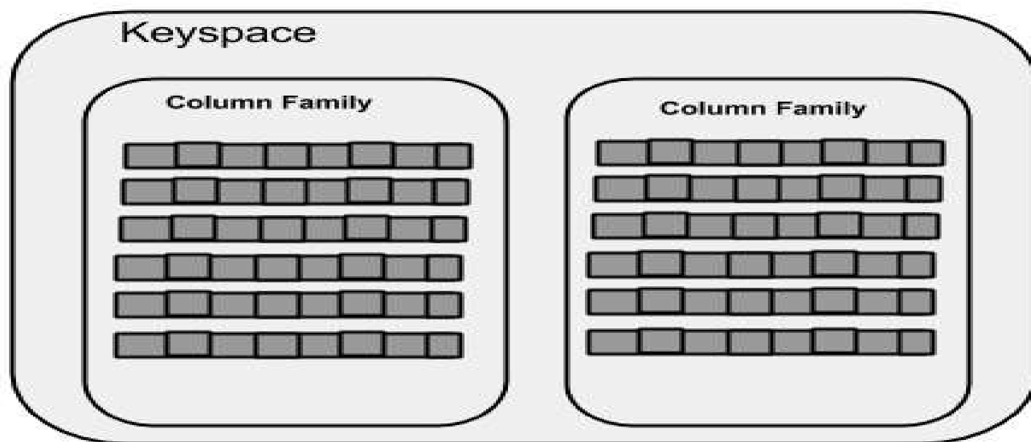
Keyspace is the outermost container for data in Cassandra. The basic attributes of a Keyspace in Cassandra are −

- **Replication factor** − It is the number of machines in the cluster that will receive copies of the same data.
- **Replica placement strategy** − It is nothing but the strategy to place replicas in the ring. We have strategies such as **simple strategy** (rack-aware strategy), **old network topology strategy** (rack-aware strategy), and **network topology strategy** (datacenter-shared strategy).
- **Column families** − Keyspace is a container for a list of one or more column families. A column family, in turn, is a container of a collection of rows. Each row contains ordered columns. Column families represent the structure of your data. Each keyspace has at least one and often many column families.

The syntax of creating a Keyspace is as follows −

CREATE KEYSPACE Keyspace name
WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};

The following illustration shows a schematic view of a Keyspace.



Column Family

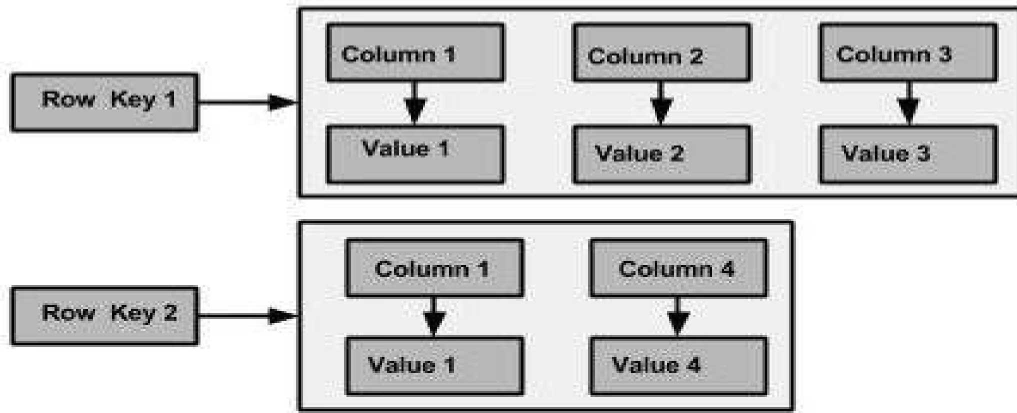
A column family is a container for an ordered collection of rows. Each row, in turn, is an ordered collection of columns. The following table lists the points that differentiate a column family from a table of relational databases.

Relational Table	Cassandra column Family
A schema in a relational model is fixed. Once we define certain columns for a table, while inserting data, in every row all the columns must be filled at least with a null value.	In Cassandra, although the column families are defined, the columns are not. You can freely add any column to any column family at any time.
Relational tables define only columns and the user fills in the table with values.	In Cassandra, a table contains columns, or can be defined as a super column family.

A Cassandra column family has the following attributes −

Note − Unlike relational tables where a column family's schema is not fixed, Cassandra does not force individual rows to have all the columns.

The following figure shows an example of a Cassandra column family.



Column

A column is the basic data structure of Cassandra with three values, namely key or column name, value, and a time stamp. Given below is the structure of a column.

Column		
name : byte[]	value : byte[]	clock : clock[]

SuperColumn

A super column is a special column; therefore, it is also a key-value pair. But a super column stores a map of sub-columns.

Generally column families are stored on disk in individual files. Therefore, to optimize performance, it is important to keep columns that you are likely to query together in the same column family, and a super column can be helpful here.Given below is the structure of a super column.

Super Column	
name : byte[]	cols : map<byte[], column>

Data Models of Cassandra and RDBMS

The following table lists down the points that differentiate the data model of Cassandra from that of an RDBMS.

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.

ii. **KEY SPACE**

Creating a Keyspace using Cqlsh

A keyspace in Cassandra is a namespace that defines data replication on nodes. A cluster contains one keyspace per node. Given below is the syntax for creating a keyspace using the statement **CREATE KEYSPACE**.

Syntax

CREATE KEYSPACE <identifier> WITH <properties>

i.e.

CREATE KEYSPACE “KeySpace Name”
WITH replication = {'class': ‘Strategy name’, 'replication_factor' : ‘No.Of replicas’};

CREATE KEYSPACE “KeySpace Name”
WITH replication = {'class': ‘Strategy name’, 'replication_factor' : ‘No.Of replicas’}

AND durable_writes = ‘Boolean value’;

The CREATE KEYSPACE statement has two properties: **replication** and **durable_writes**.

Replication

The replication option is to specify the **Replica Placement strategy** and the number of replicas wanted. The following table lists all the replica placement strategies.

Strategy name	Description
Simple Strategy'	Specifies a simple replication factor for the cluster.

	data-center independently.
Old Network Topology Strategy	This is a legacy replication strategy.

Using this option, you can instruct Cassandra whether to use **commitlog** for updates on the current KeySpace. This option is not mandatory and by default, it is set to true.

Example

Given below is an example of creating a KeySpace.

- Here we are creating a KeySpace named **TutorialsPoint**.
- We are using the first replica placement strategy, i.e.., **Simple Strategy**.
- And we are choosing the replication factor to **1 replica**.

```
cqlsh.> CREATE KEYSPACE tutorialspoint
WITH replication = {'class':'SimpleStrategy', 'replication_factor' : 3};
```

Verification

You can verify whether the table is created or not using the command **Describe**. If you use this command over keyspaces, it will display all the keyspaces created as shown below.

```
cqlsh> DESCRIBE keyspaces;

tutorialspoint system system_traces
```

Here you can observe the newly created KeySpace **tutorialspoint.**

Durable_writes

By default, the durable_writes properties of a table is set to **true,** however it can be set to false. You cannot set this property to **simplex strategy.**

Example

Given below is the example demonstrating the usage of durable writes property.

```
cqlsh> CREATE KEYSPACE test
... WITH REPLICATION = { 'class' : 'NetworkTopologyStrategy', 'datacenter1' : 3 }
... AND DURABLE_WRITES = false;
```

Verification

You can verify whether the durable_writes property of test KeySpace was set to false by querying the System Keyspace. This query gives you all the KeySpaces along with their properties.

```
cqlsh> SELECT * FROM system_schema.keyspaces;

 keyspace_name | durable_writes |                          strategy_class | strategy_options
----------------+----------------+------------------------------------------------------+----------------------------


       test |        False | org.apache.cassandra.locator.NetworkTopologyStrategy | {"datacenter1" :
"3"}

 tutorialspoint |         True |        org.apache.cassandra.locator.SimpleStrategy | {"replication_factor"
: "4"}
```

{"replication_factor" : "2"}

(4 rows)

Here you can observe the durable_writes property of test KeySpace was set to false.

Using a Keyspace

You can use a created KeySpace using the keyword **USE**. Its syntax is as follows −

Syntax:USE <identifier>

Example

In the following example, we are using the KeySpace **tutorialspoint.**

```
cqlsh> USE tutorialspoint;
cqlsh:tutorialspoint>
```

Creating a Keyspace using Java API

You can create a Keyspace using the **execute()** method of **Session** class. Follow the steps given below to create a keyspace using Java API.

Step1: Create a Cluster Object

First of all, create an instance of **Cluster.builder** class of **com.datastax.driver.core** package as shown below.

```
//Creating Cluster.Builder object

Cluster.Builder builder1 = Cluster.builder();
```

Add a contact point (IP address of the node) using **addContactPoint()** method of **Cluster.Builder** object. This method returns **Cluster.Builder**.

```
//Adding contact point to the Cluster.Builder object

Cluster.Builder builder2 = build.addContactPoint( "127.0.0.1" );
```

Using the new builder object, create a cluster object. To do so, you have a method called **build()** in the **Cluster.Builder** class. The following code shows how to create a cluster object.

```
//Building a cluster
Cluster cluster = builder.build();
```

You can build a cluster object in a single line of code as shown below.

```
Cluster cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
```

Step 2: Create a Session Object

Create an instance of **Session** object using the **connect()** method of **Cluster** class as shown below.

```
Session session = cluster.connect( );
```

This method creates a new session and initializes it. If you already have a keyspace, you can set it to the existing one by passing the keyspace name in string format to this method as shown below.

```
Session session = cluster.connect(" Your keyspace name " );
```

Step 3: Execute Query

You can execute **CQL** queries using the **execute()** method of **Session** class. Pass the query either in

strategy, i.e., Simple Strategy, and we are choosing the replication factor to 1 replica.

You have to store the query in a string variable and pass it to the execute() method as shown below.

```
String query = "CREATE KEYSPACE tp WITH replication "
  + "= {'class':'SimpleStrategy', 'replication_factor':1}; ";
session.execute(query);
```

Step4 : Use the KeySpace

You can use a created KeySpace using the execute() method as shown below.

execute(“ USE tp ”);

Given below is the complete program to create and use a keyspace in Cassandra using Java API.

```
import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.Session;

public class Create_KeySpace {

  public static void main(String args[]){

    //Query
    String query = "CREATE KEYSPACE tp WITH replication "
      + "= {'class':'SimpleStrategy', 'replication_factor':1};";

    //creating Cluster object
    Cluster cluster = Cluster.builder().addContactPoint("127.0.0.1").build();

    //Creating Session object
    Session session = cluster.connect();

    //Executing the query
    session.execute(query);

    //using the KeySpace
    session.execute("USE tp");
    System.out.println("Keyspace created");
  }
}
```

Save the above program with the class name followed by .java, browse to the location where it is saved. Compile and execute the program as shown below.

```
$javac Create_KeySpace.java
$java Create_KeySpace
```

Under normal conditions, it will produce the following output −

Keyspace created

iii. Table Operations

Cassandra is a distributed NoSQL database system. It offers high scalability, availability, and fault-tolerance. It uses a decentralized architecture, where data is distributed across multiple nodes, and

column has name, data type, and an optional value. To create a table, you need to specify the keyspace. It is namespace that defines the replication strategy, and the table name. You also need to specify the columns and their data types.

Example-1

```
CREATE TABLE example_table (
  id int PRIMARY KEY,
  name text,
  age int
);
```

Example-2

```
CREATE TABLE sample_table (
  id INT PRIMARY KEY,
  name TEXT,
  age INT,
  email TEXT
);
```

Inserting Data

After defining and creating a dabase table. You can insert data into it. To insert data, You need to specify keyspace, table name, and values for each column. You can use the NULL keyword when a column has no value.

Example-1

```
INSERT INTO example_table (id, name, age) VALUES (1, 'John', 30);
```

Example-2

```
INSERT INTO sample_table (id, name, age, email) VALUES (1, 'John Doe', 35,
'johndoe@example.com');
INSERT INTO sample_table (id, name, age, email) VALUES (2, 'Jane Doe', 30,
'janedoe@example.com');
INSERT INTO sample_table (id, name, age, email) VALUES (3, 'Bob Smith', 45,
'bobsmith@example.com');
```

Updating Data

To update data, you can use the UPDATE statement. You need to specify keyspace, table name, columns to update, and new values. You also need to specify the WHERE clause to identify the row to update.

Example-1

```
UPDATE example_table SET age = 35 WHERE id = 1;
```

Example-2

```
UPDATE sample_table SET age = 40 WHERE id = 2;
```

Example-1

```
DELETE FROM example_table WHERE id = 1;
```

Example-2

```
DELETE FROM sample_table WHERE id = 3;
```

Selecting Data

You can use the SELECT statement to select data. You need to specify keyspace, table name, and columns to retrieve. You also need to specify the WHERE clause to filter the rows. You can use the ORDER BY clause to sort the rows. The LIMIT clause to limit the number of rows returned.

Example-1

```
SELECT name, age FROM example_table WHERE age > 25 ORDER BY age DESC LIMIT 10;
```

Example-2

```
SELECT * FROM sample_table WHERE age > 30;

SELECT name, email FROM sample_table WHERE id = 1;
```

ALTER TABLE

To modify an existing table structure in Cassandra, you can use the ALTER TABLE statement.

Example

```
ALTER TABLE users

ADD surname TEXT;
```

It adds a new column named 'surname' of type 'TEXT' to the 'users' table.

DROP TABLE

To delete an existing table in Cassandra. You can use the DROP TABLE statement.

Example

```
DROP TABLE users;
```

<u>Iv.CRUD operations</u>

Cassandra CRUD Operation stands for Create, Update, Read and Delete or Drop. These operations are used to manipulate data in Cassandra. Apart from this, CRUD operations in Cassandra, a user can also verify the command or the data.

a. Create Operation

A user can insert data into the table using Cassandra CRUD operation. The data is stored in the columns of a row in the table. Using INSERT command with proper what, a user can perform this operation.

<u>Read about Important Features of Cassandra</u>

A Syntax of Create Operation-

INSERT INTO <table name>

(<column1>,<column2>....)

VALUES (<value1>,<value2>...)

USING<option>

EN	NAME	BRANCH	PHONE	CITY
001	Ayush	Electrical Engineering	9999999999	Boston
002	Aarav	Computer Engineering	8888888888	New York C
003	Kabir	Applied Physics	7777777777	Philadelphi

EXAMPLE 1: Creating a table and inserting the data into a table:
INPUT:

cqlsh:keyspace1> INSERT INTO student(en, name, branch, phone, city)

VALUES(001, 'Ayush', 'Electrical Engineering', 9999999999, 'Boston');

cqlsh:keyspace1> INSERT INTO student(en, name, branch, phone, city)

VALUES(002, 'Aarav', 'Computer Engineering', 8888888888, 'New York City');

cqlsh:keyspace1> INSERT INTO student(en, name, branch, phone, city)

VALUES(003, 'Kabir', 'Applied Physics', 7777777777, 'Philadelphia');

Let's Learn Cassandra Architecture in detail

Table.2 Cassandra Crud Operation – OUTPUT After Verification (READ operation)

EN	NAME	BRANCH	PHONE	CITY
001	Ayush	Electrical Engineering	9999999999	Boston
002	Aarav	Computer Engineering	8888888888	New York C
003	Kabir	Applied Physics	7777777777	Philadelphi

b.Update Operation

The second operation in the Cassandra CRUD operation is the UPDATE operation. A user can use UPDATE command for the operation. This operation uses three keywords while updating the table.

- **Where:** This keyword will specify the location where data is to be updated.
- **Set:** This keyword will specify the updated value.
- **Must:** This keyword includes the columns composing the primary key.

Furthermore, at the time of updating the rows, if a row is unavailable, then Cassandra has a feature to create a fresh row for the same.

Do you know How Cassandra Stores Data

A Syntax of Update Operation-

UPDATE <table name>

WHERE <condition>

EXAMPLE 2: Let's change few details in the table 'student'. In this example, we will update Aarav's city from 'New York City' to 'San Fransisco'.
INPUT:
cqlsh:keyspace1> UPDATE student SET city='San Fransisco'

WHERE en=002;

Table.3 Cassandra Crud Operation – OUTPUT After Verification

EN	NAME	BRANCH	PHONE	CITY
001	Ayush	Electrical Engineering	9999999999	Boston
002	Aarav	Computer Engineering	8888888888	San Fransis
003	Kabir	Applied Physics	7777777777	Philadelphi

c. Read Operation

This is the third Cassandra CRUD Operation – Read Operation. A user has a choice to read either the whole table or a single column. To read data from a table, a user can use SELECT clause. This command is also used for verifying the table after every operation.

Have a look at Cassandra Shell Commands

SYNTAX to read the whole table-

SELECT * FROM <table name>;

EXAMPLE 3: To read the whole table 'student'.
INPUT:
cqlsh:keyspace1> SELECT * FROM student;

Table.4 Cassandra Crud Operation – OUTPUT After Verification

EN	NAME	BRANCH	PHONE	CITY
001	Ayush	Electrical Engineering	9999999999	Boston
002	Aarav	Computer Engineering	8888888888	San Fransis
003	Kabir	Applied Physics	7777777777	Philadelphi

SYNTAX to read selected columns-
SELECT <column name1>,<column name2>.... FROM <table name>;

EXAMPLE 4: To read columns of name and city from table 'student'.
INPUT:

Read More about Why Cassandra is So Popular?

Table.5 Cassandra Crud Operation – OUTPUT After Verification

NAME	CITY
Ayush	Boston
Aarav	San Fransisco
Kabir	Philadelphia

d. Delete Operation

Delete operation is the last Cassandra CRUD Operation, allows a user to delete data from a table. The user can use DELETE command for this operation.

A Syntax of Delete Operation-

DELETE <identifier> FROM <table name> WHERE <condition>;

EXAMPLE 5: In the 'student' table let us delete the 'phone' or phone number from 003 row.

cqlsh:keyspace1> DELETE phone FROM student WHERE en=003;

Table.6 Cassandra Crud Operation – OUTPUT After Verification

EN	NAME	BRANCH	PHONE	CITY
001	Ayush	Electrical Engineering	9999999999	Boston
002	Aarav	Computer Engineering	8888888888	San Fransis
003	Kabir	Applied Physics	null	Philadelphi

SYNTAX for deleting the entire row-

DELETE FROM <identifier> WHERE <condition>;

EXAMPLE 6: In the 'student' table, let us delete the entire third row.
cqlsh:keyspace1> DELETE FROM student WHERE en=003;

Let's Explore Best Books To Learn Cassandra

Table.7 Cassandra Crud Operation – OUTPUT After Verification

EN	NAME	BRANCH	PHONE	CITY
001	Ayush	Electrical Engineering	9999999999	Boston
002	Aarav	Computer Engineering	8888888888	San Fransisco

CQL provides a rich set of built-in data types, including collection types. Along with these data types, users can also create their own custom data types. The following table provides a list of built-in data types available in CQL.

Data Type	Constants	Description
ascii	strings	Represents ASCII character string
bigint	bigint	Represents 64-bit signed long
blob	blobs	Represents arbitrary bytes
Boolean	booleans	Represents true or false
counter	integers	Represents counter column
decimal	integers, floats	Represents variable-precision decimal
double	integers	Represents 64-bit IEEE-754 floating point
float	integers, floats	Represents 32-bit IEEE-754 floating point
inet	strings	Represents an IP address, IPv4 or IPv6
int	integers	Represents 32-bit signed int
text	strings	Represents UTF8 encoded string
timestamp	integers, strings	Represents a timestamp
timeuuid	uuids	Represents type 1 UUID
uuid	uuids	Represents type 1 or type 4
		UUID
varchar	strings	Represents uTF8 encoded string

Cassandra Query Language also provides a collection data types. The following table provides a list of Collections available in CQL.

Collection	Description
list	A list is a collection of one or more ordered elements.
map	A map is a collection of key-value pairs.
set	A set is a collection of one or more elements.

User-defined datatypes

Cqlsh provides users a facility of creating their own data types. Given below are the commands used while dealing with user defined datatypes.

- **CREATE TYPE** − Creates a user-defined datatype.
- **ALTER TYPE** − Modifies a user-defined datatype.
- **DROP TYPE** − Drops a user-defined datatype.
- **DESCRIBE TYPE** − Describes a user-defined datatype.
- **DESCRIBE TYPES** − Describes user-defined datatypes.

3.5 HIVE: DATA TYPES, DATABASE OPERATIONS, PARTITIONING – HIVEQL

Hive is a data warehouse system which is used to analyze structured data. It is built on the top of Hadoop. It was developed by Facebook. Hive provides the functionality of reading, writing, and managing large datasets residing in distributed storage.

i. DATA TYPES

his chapter takes you through the different data types in Hive, which are involved in the table creation. All the data types in Hive are classified into four types, given as follows:

- Column Types
- Literals
- Null Values
- Complex Types

Column Types

Column types are used as column data types of Hive. They are as follows:

Integral Types

Integer type data can be specified using integral data types, INT. When the data range exceeds the range of INT, you need to use BIGINT and if the data range is smaller than the INT, you use SMALLINT. TINYINT is smaller than SMALLINT.

The following table depicts various INT data types:

Type	Postfix	Example

SMALLINT	S	10S
INT	-	10
BIGINT	L	10L

String Types

String type data types can be specified using single quotes (' ') or double quotes (" "). It contains two data types: VARCHAR and CHAR. Hive follows C-types escape characters.

The following table depicts various CHAR data types:

Data Type	Length
VARCHAR	1 to 65355
CHAR	255

Timestamp

It supports traditional UNIX timestamp with optional nanosecond precision. It supports java.sql.Timestamp format “YYYY-MM-DD HH:MM:SS.fffffffff” and format “yyyy-mm-dd hh:mm:ss.ffffffffff”.

Dates

DATE values are described in year/month/day format in the form {{YYYY-MM-DD}}.

Decimals

The DECIMAL type in Hive is as same as Big Decimal format of Java. It is used for representing immutable arbitrary precision. The syntax and example is as follows:

```
DECIMAL(precision, scale)
decimal(10,0)
```

Union Types

Union is a collection of heterogeneous data types. You can create an instance using **create union**. The syntax and example is as follows:

```
UNIONTYPE<int, double, array<string>, struct<a:int,b:string>>

{0:1}
{1:2.0}
{2:["three","four"]}
{3:{"a":5,"b":"five"}}
{2:["six","seven"]}
{3:{"a":8,"b":"eight"}}
{0:9}
```

Floating Point Types

Floating point types are nothing but numbers with decimal points. Generally, this type of data is composed of DOUBLE data type.

Decimal Type

Decimal type data is nothing but floating point value with higher range than DOUBLE data type. The range of decimal type is approximately -10^{-308} to 10^{308}.

Null Value

Missing values are represented by the special value NULL.

Complex Types

The Hive complex data types are as follows:

Arrays

Arrays in Hive are used the same way they are used in Java.

```
Syntax: ARRAY<data_type>
```

Maps

Maps in Hive are similar to Java Maps.

```
Syntax: MAP<primitive_type, data_type>
```

Structs

Structs in Hive is similar to using complex data with comment.

```
Syntax: STRUCT<col_name : data_type [COMMENT col_comment], ...>
```

ii.DATABASE OPERATIONS

Hive is a database technology that can define databases and tables to analyze structured data. The theme for structured data analysis is to store the data in a tabular manner, and pass queries to analyze it. This chapter explains how to create Hive database. Hive contains a default database named **default**.

Create Database Statement

Create Database is a statement used to create a database in Hive. A database in Hive is a **namespace** or a collection of tables. The **syntax** for this statement is as follows:

```
CREATE DATABASE|SCHEMA [IF NOT EXISTS] <database name>
```

Here, IF NOT EXISTS is an optional clause, which notifies the user that a database with the same name already exists. We can use SCHEMA in place of DATABASE in this command. The following query is executed to create a database named **userdb**:

```
hive> CREATE DATABASE [IF NOT EXISTS] userdb;
```

or

```
hive> CREATE SCHEMA userdb;
```

The following query is used to verify a databases list:

JDBC Program

The JDBC program to create a database is given below.

```
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.DriverManager;

public class HiveCreateDb {
  private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

  public static void main(String[] args) throws SQLException {
    // Register driver and create driver instance

    Class.forName(driverName);
    // get connection

    Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/default", "", "");
    Statement stmt = con.createStatement();

    stmt.executeQuery("CREATE DATABASE userdb");
    System.out.println("Database userdb created successfully.");

    con.close();
  }
}
```

Save the program in a file named HiveCreateDb.java. The following commands are used to compile and execute this program.

```
$ javac HiveCreateDb.java
$ java HiveCreateDb
```

Output:

Database userdb created successfully.

Drop Database Statement

Drop Database is a statement that drops all the tables and deletes the database. Its syntax is as follows:

```
DROP DATABASE StatementDROP (DATABASE|SCHEMA) [IF EXISTS] database_name
[RESTRICT|CASCADE];
```

The following queries are used to drop a database. Let us assume that the database name is **userdb**.

```
hive> DROP DATABASE IF EXISTS userdb;
```

The following query drops the database using **CASCADE**. It means dropping respective tables before dropping the database.

```
hive> DROP DATABASE IF EXISTS userdb CASCADE;
```

The following query drops the database using **SCHEMA**.

JDBC Program

The JDBC program to drop a database is given below.

```
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.DriverManager;

public class HiveDropDb {
  private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

  public static void main(String[] args) throws SQLException {

    // Register driver and create driver instance
    Class.forName(driverName);

    // get connection
    Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/default", "", "");
    Statement stmt = con.createStatement();
    stmt.executeQuery("DROP DATABASE userdb");

    System.out.println("Drop userdb database successful.");

    con.close();
  }
}
```

Save the program in a file named HiveDropDb.java. Given below are the commands to compile and execute this program.

```
$ javac HiveDropDb.java
$ java HiveDropDb
```

Output:

Drop userdb database successful.

Database Operations in HIVE

1. Create a database

Syntax:

create database database_name;

Example:

create database geeksportal;

Output:

```
hive> create database geeksportal;
OK
Time taken: 0.414 seconds
hive>
```

Example:
create table geeksportal.geekdata(id int,name string);

Here id and string are the two columns.

Output :

```
hive> create table geeksportal.geekdata(id int,name string);
OK
Time taken: 0.289 seconds
```

3. Display Database
Syntax:
show databases;

Output: Display the databases created.

```
hive> show databases;
OK
default
demo
exp12
geeksportal
sravan1
Time taken: 0.648 seconds, Fetched: 5 row(s)
hive>
```

4. Describe Database
Syntax:
describe database database_name;

Example:
describe database geeksportal;

Output: Display the HDFS path of a particular database.

```
hive> describe database geeksportal;
OK
geeksportal          hdfs://quickstart.cloudera:8020/user/hive/warehouse/geeksport
Time taken: 0.053 seconds, Fetched: 1 row(s)
hive>
```

iii. PARTITIONING

Hive organizes tables into partitions. It is a way of dividing a table into related parts based on the values of partitioned columns such as date, city, and department. Using partition, it is easy to query a portion of the data.

Tables or partitions are sub-divided into **buckets,** to provide extra structure to the data that may be used for more efficient querying. Bucketing works based on the value of hash function of some column of a table.

For example, a table named **Tab1** contains employee data such as id, name, dept, and yoj (i.e., year of joining). Suppose you need to retrieve the details of all employees who joined in 2012. A query searches the whole table for the required information. However, if you partition the employee data with the year and store it in a separate file, it reduces the query processing time. The following example shows how to partition a file and its data:

Id, name, dept, yoj
1, gopal, TP, 2012
2, kiran, HR, 2012
3, kaleel,SC, 2013
4, Prasanth, SC, 2013

The above data is partitioned into two files using year.

/tab1/employeedata/2012/file2

1, gopal, TP, 2012
2, kiran, HR, 2012

/tab1/employeedata/2013/file3

3, kaleel,SC, 2013
4, Prasanth, SC, 2013

Adding a Partition

We can add partitions to a table by altering the table. Let us assume we have a table called **employee** with fields such as Id, Name, Salary, Designation, Dept, and yoj.

Syntax:

```
ALTER TABLE table_name ADD [IF NOT EXISTS] PARTITION partition_spec
[LOCATION 'location1'] partition_spec [LOCATION 'location2'] ...;

partition_spec:
: (p_column = p_col_value, p_column = p_col_value, ...)
```

The following query is used to add a partition to the employee table.

```
hive> ALTER TABLE employee
> ADD PARTITION (year='2012')
> location '/2012/part2012';
```

Renaming a Partition

The syntax of this command is as follows.

```
ALTER TABLE table_name PARTITION partition_spec RENAME TO PARTITION partition_spec;
```

The following query is used to rename a partition:

```
hive> ALTER TABLE employee PARTITION (year='1203')
  > RENAME TO PARTITION (Yoj='1203');
```

Dropping a Partition

The following syntax is used to drop a partition:

```
ALTER TABLE table_name DROP [IF EXISTS] PARTITION partition_spec, PARTITION
partition_spec,...;
```

The following query is used to drop a partition:

```
hive> ALTER TABLE employee DROP [IF EXISTS]
  > PARTITION (year='1203');
```

iv.HIVEQL

SELECT statement is used to retrieve the data from a table. WHERE clause works similar to a condition. It filters the data using the condition and gives you a finite result. The built-in operators and functions generate an expression, which fulfils the condition.

Syntax

Given below is the syntax of the SELECT query:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having_condition]
[CLUSTER BY col_list | [DISTRIBUTE BY col_list] [SORT BY col_list]]
[LIMIT number];
```

Example

Let us take an example for SELECT…WHERE clause. Assume we have the employee table as given below, with fields named Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details who earn a salary of more than Rs 30000.

```
+------+--------------+-------------+-------------------+--------+
| ID   | Name         | Salary      | Designation       | Dept   |
+------+--------------+-------------+-------------------+--------+
|1201  | Gopal        | 45000       | Technical manager | TP     |
|1202  | Manisha      | 45000       | Proofreader       | PR     |
|1203  | Masthanvali  | 40000       | Technical writer  | TP     |
|1204  | Krian        | 40000       | Hr Admin          | HR     |
|1205  | Kranthi      | 30000       | Op Admin          | Admin  |
+------+--------------+-------------+-------------------+--------+
```

The following query retrieves the employee details using the above scenario:

```
hive> SELECT * FROM employee WHERE salary>30000;
```

On successful execution of the query, you get to see the following response:

```
+------+--------------+-------------+-------------------+--------+
| ID   | Name         | Salary      | Designation       | Dept   |
+------+--------------+-------------+-------------------+--------+
|1201  | Gopal        | 45000       | Technical manager | TP     |
|1202  | Manisha      | 45000       | Proofreader       | PR     |
|1203  | Masthanvali  | 40000       | Technical writer  | TP     |
|1204  | Krian        | 40000       | Hr Admin          | HR     |
+------+--------------+-------------+-------------------+--------+
```

JDBC Program

The JDBC program to apply where clause for the given example is as follows.

```
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.DriverManager;
```

```
  public static void main(String[] args) throws SQLException {

    // Register driver and create driver instance
    Class.forName(driverName);

    // get connection
    Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "", "");

    // create statement
    Statement stmt = con.createStatement();

    // execute statement
    Resultset res = stmt.executeQuery("SELECT * FROM employee WHERE salary>30000;");

    System.out.println("Result:");
    System.out.println(" ID \t Name \t Salary \t Designation \t Dept ");

    while (res.next()) {
      System.out.println(res.getInt(1) + " " + res.getString(2) + " " + res.getDouble(3) + " " +
res.getString(4) + " " + res.getString(5));
    }
    con.close();
  }
}
```

Save the program in a file named HiveQLWhere.java. Use the following commands to compile and execute this program.

```
$ javac HiveQLWhere.java
$ java HiveQLWhere
```

Output:

ID	Name	Salary	Designation	Dept
1201	Gopal	45000	Technical manager	TP
1202	Manisha	45000	Proofreader	PR
1203	Masthanvali	40000	Technical writer	TP
1204	Krian	40000	Hr Admin	HR

3.6 ORIENTDB GRAPH DATABASE – ORIENTDB FEATURES

n graph databases, the database system graphs data into network-like structures consisting of vertices and edges. In the OrientDB Graph model, the database represents data through the concept of a property graph, which defines a vertex as an entity linked with other vertices and an edge, as an entity that links two vertices.

OrientDB ships with a generic vertex persistent class, called V, as well as a class for edges, called E. As an example, you can create a new vertex using the INSERT command with V.

orientdb> INSERT INTO V SET name='Jay'

Created record with RID #9:0

orientdb> CREATE VERTEX V SET name='Jay'

Created vertex with RID #9:1

By using the graph commands over the standard SQL syntax, OrientDB ensures that your graphs remain consistent. For more information on the particular commands, see the following pages:

- CREATE VERTEX
- DELETE VERTEX
- CREATE EDGE
- UPDATE EDGE
- DELETE EDGE

Use Case: Social Network for Restaurant Patrons

While you have the option of working with vertexes and edges in your database as they are, you can also extend the standard V and E classes to suit the particular needs of your application. The advantages of this approach are,

- It grants better understanding about the meaning of these entities.
- It allows for optional constraints at the class level.
- It improves performance through better partitioning of entities.
- It allows for object-oriented inheritance among the graph elements.

For example, consider a social network based on restaurants. You need to start with a class for individual customers and another for the restaurants they patronize. Create these classes to extend the V class.

orientdb> CREATE CLASS Person EXTENDS V

orientdb> CREATE CLASS Restaurant EXTENDS V

Doing this creates the schema for your social network. Now that the schema is ready, populate the graph with data.

orientdb> CREATE VERTEX Person SET name='Luca'

Created record with RID #11:0

orientdb> CREATE VERTEX Person SET name='Bill'

Created record with RID #11:1

orientdb> CREATE VERTEX Person SET name='Jay'

orientdb> CREATE VERTEX Restaurant SET name='Dante', type='Pizza'

Created record with RID #12:0

orientdb> CREATE VERTEX Restaurant SET name='Charlie', type='French'

Created record with RID #12:1

This adds three vertices to the Person class, representing individual users in the social network. It also adds two vertices to the Restaurant class, representing the restaurants that they patronize.

Creating Edges

For the moment, these vertices are independent of one another, tied together only by the classes to which they belong. That is, they are not yet connected by edges. Before you can make these connections, you first need to create a class that extends E.

orientdb> CREATE CLASS Eat EXTENDS E

This creates the class Eat, which extends the class E. Eat represents the relationship between the vertex Person and the vertex Restaurant.

When you create the edge from this class, note that the orientation of the vertices is important, because it gives the relationship its meaning. For instance, creating an edge in the opposite direction, (from Restaurant to Person), would call for a separate class, such as Attendee.

The user Luca eats at the pizza joint Dante. Create an edge that represents this connection:

```
orientdb> CREATE EDGE Eat FROM ( SELECT FROM Person WHERE name='Luca' )
        TO ( SELECT FROM Restaurant WHERE name='Dante' )
```

Creating Edges from Record ID

In the event that you know the Record ID of the vertices, you can connect them directly with a shorter and faster command. For example, the person Bill also eats at the restaurant Dante and the person Jay eats at the restaurant Charlie. Create edges in the class Eat to represent these connections.

orientdb> CREATE EDGE Eat FROM #11:1 TO #12:0

orientdb> CREATE EDGE Eat FROM #11:2 TO #12:1

Querying Graphs

In the above example you created and populated a small graph of a social network of individual users and the restaurants at which they eat. You can now begin to experiment with queries on a graph database.

BOTH() To retrieve the adjacent incoming and outgoing vertices

For example, to know all of the people who eat in the restaurant Dante, which has a Record ID of #12:0, you can access the record for that restaurant and traverse the incoming edges to discover which entries in the Person class connect to it.

orientdb> SELECT IN() FROM Restaurant WHERE name='Dante'

```
-------+----------------+
 @RID  | in            |
-------+----------------+
 #-2:1 | [#11:0, #11:1] |
-------+----------------+
```

This query displays the record ID's from the Person class that connect to the restaurant Dante. In cases such as this, you can use the EXPAND() special function to transform the vertex collection in the result-set by expanding it.

orientdb> SELECT EXPAND(IN()) FROM Restaurant WHERE name='Dante'

```
-------+-------------+-------------+---------+
 @RID  | @CLASS      | Name        | out_Eat |
-------+-------------+-------------+---------+
 #11:0 | Person      | Luca        | #12:0   |
 #11:1 | Person      | Bill        | #12:0   |
-------+-------------+-------------+---------+
```

Creating Edge to Connect Users

Your application at this point shows connections between individual users and the restaurants they patronize. While this is interesting, it does not yet function as a social network. To do so, you need to establish edges that connect the users to one another.

To begin, as before, create a new class that extends E:

orientdb> CREATE CLASS Friend EXTENDS E

The users Luca and Jay are friends. They have Record ID's of #11:0 and #11:2. Create an edge that connects them.

orientdb> CREATE EDGE Friend FROM #11:0 TO #11:2

In the Friend relationship, orientation is not important. That is, if Luca is a friend of Jay's then Jay is a friend of Luca's. Therefore, you should use the BOTH() function.

orientdb> SELECT EXPAND(BOTH('Friend')) FROM Person WHERE name = 'Luca'

```
-------+-------------+-------------+---------+-----------+
 @RID  | @CLASS      | Name        | out_Eat | in_Friend |
```

Here, the BOTH() function takes the edge class Friend as an argument, crossing only relationships of the Friend kind, (that is, it skips the Eat class, at this time). Note in the result-set that the relationship with Luca, with a Record ID of #11:0 in the in_ field.

You can also now view all the restaurants patronized by friends of Luca.

```
orientdb> SELECT EXPAND( BOTH('Friend').out('Eat') ) FROM Person
        WHERE name='Luca'
```

@RID	@CLASS	Name	Type	in_Eat
#12:1	Restaurant	Charlie	French	#11:2

Lightweight Edges

In version 1.4.x, OrientDB begins to manage some edges as Lightweight Edges. Lightweight Edges do not have Record ID's, but are physically stored as links within vertices. Note that OrientDB only uses a Lightweight Edge only when the edge has no properties, otherwise it uses the standard Edge.

From the logic point of view, Lightweight Edges are Edges in all effects, so that all graph functions work with them. This is to improve performance and reduce disk space.

Because Lightweight Edges don't exist as separate records in the database, some queries won't work as expected. For instance,

```
orientdb> SELECT FROM E
```

For most cases, an edge is used connecting vertices, so this query would not cause any problems in particular. But, it would not return Lightweight Edges in the result-set. In the event that you need to query edges directly, including those with no properties, disable the Lightweight Edge feature.

To disable the Lightweight Edge feature, execute the following command.

```
orientdb> ALTER DATABASE CUSTOM useLightweightEdges=FALSE
```

You only need to execute this command once. OrientDB now generates new edges as the standard Edge, rather than the Lightweight Edge. Note that this does not affect existing edges.

For troubleshooting information on Lightweight Edges, see Why I can't see all the edges. For more information in the Graph model in OrientDB, see Graph API.

i. ORIENTDB FEATURES

The main feature of OrientDB is to support multi-model objects, i.e. it supports different models like Document, Graph, Key/Value and Real Object. It contains a separate API to support all these four models.

The terminology Document model belongs to NoSQL database. It means the data is stored in the Documents and the group of Documents are called as **Collection**. Technically, document means a set of key/value pairs or also referred to as fields or properties.

OrientDB uses the concepts such as classes, clusters, and link for storing, grouping, and analyzing the documents.

The following table illustrates the comparison between relational model, document model, and OrientDB document model −

Relational Model	Document Model	OrientDB Document Model
Table	Collection	Class or Cluster
Row	Document	Document
Column	Key/value pair	Document field
Relationship	Not available	Link

Graph Model

A graph data structure is a data model that can store data in the form of Vertices (Nodes) interconnected by Edges (Arcs). The idea of OrientDB graph database came from property graph. The vertex and edge are the main artifacts of the Graph model. They contain the properties, which can make these appear similar to documents.

The following table shows a comparison between graph model, relational data model, and OrientDB graph model.

Relational Model	Graph Model	OrientDB Graph Model
Table	Vertex and Edge Class	Class that extends "V" (for Vertex) and "E" (for Edges)
Row	Vertex	Vertex
Column	Vertex and Edge property	Vertex and Edge property
Relationship	Edge	Edge

The Key/Value Model

The Key/Value model means that data can be stored in the form of key/value pair where the values can be of simple and complex types. It can support documents and graph elements as values.

Relational Model	Key/Value Model	OrientDB Key/Value Model
Table	Bucket	Class or Cluster
Row	Key/Value pair	Document
Column	Not available	Document field or Vertex/Edge property
Relationship	Not available	Link

The Object Model

This model has been inherited by Object Oriented programming and supports **Inheritance** between types (sub-types extends the super-types), **Polymorphism** when you refer to a base class and **Direct binding** from/to Objects used in programming languages.

The following table illustrates the comparison between relational model, Object model, and OrientDB Object model.

Relational Model	Object Model	OrientDB Object Model
Table	Class	Class or Cluster
Row	Object	Document or Vertex
Column	Object property	Document field or Vertex/Edge property
Relationship	Pointer	Link

Before go ahead in detail, it is better to know the basic terminology associated with OrientDB. Following are some of the important terminologies.

Record

The smallest unit that you can load from and store in the database. Records can be stored in four types.

- Document
- Record Bytes
- Vertex
- Edge

Record ID

When OrientDB generates a record, the database server automatically assigns a unit identifier to the record, called RecordID (RID). The RID looks like #<cluster>:<position>. <cluster> means cluster identification number and the <position> means absolute position of the record in the cluster.

The Document is the most flexible record type available in OrientDB. Documents are softly typed and are defined by schema classes with defined constraint, but you can also insert the document without any schema, i.e. it supports schema-less mode too.

Documents can be easily handled by export and import in JSON format. For example, take a look at the following JSON sample document. It defines the document details.

```
{
  "id"       : "1201",
  "name"     : "Jay",
  "job"      : "Developer",
  "creations" : [
    {
      "name"    : "Amiga",
      "company" : "Commodore Inc."
    },

    {
      "name"    : "Amiga 500",
      "company" : "Commodore Inc."
    }
  ]
}
```

RecordBytes

Record Type is the same as BLOB type in RDBMS. OrientDB can load and store document Record type along with binary data.

Vertex

OrientDB database is not only a Document database but also a Graph database. The new concepts such as Vertex and Edge are used to store the data in the form of graph. In graph databases, the most basic unit of data is node, which in OrientDB is called a vertex. The Vertex stores information for the database.

Edge

There is a separate record type called the Edge that connects one vertex to another. Edges are bidirectional and can only connect two vertices. There are two types of edges in OrientDB, one is regular and another one lightweight.

Class

The class is a type of data model and the concept drawn from the Object-oriented programming paradigm. Based on the traditional document database model, data is stored in the form of collection, while in the Relational database model data is stored in tables. OrientDB follows the Document API along with OPPS paradigm. As a concept, the class in OrientDB has the closest relationship with the table in relational databases, but (unlike tables) classes can be schema-less, schema-full or mixed. Classes can inherit from other classes, creating trees of classes. Each class has its own cluster or clusters, (created by default, if none are defined).

Cluster

Cluster is an important concept which is used to store records, documents, or vertices. In simple words, Cluster is a place where a group of records are stored. By default, OrientDB will create one

The CREATE class is a command used to create a cluster with specific name. Once the cluster is created you can use the cluster to save records by specifying the name during the creation of any data model.

Relationships

OrientDB supports two kinds of relationships: referenced and embedded. **Referenced relationships** means it stores direct link to the target objects of the relationships. **Embedded relationships** means it stores the relationship within the record that embeds it. This relationship is stronger than the reference relationship.

Database

The database is an interface to access the real storage. IT understands high-level concepts such as queries, schemas, metadata, indices, and so on. OrientDB also provides multiple database types. For more information on these types, see Database Types.

UNIT – IV XML DATABASES

STRUCTURED, SEMI STRUCTURED AND UNSTRUCTURED DATA

What Is Data?

- Data is a set of facts such as descriptions, observations, and numbers used in decision making.
- We can classify data as structured, unstructured, or semi-structured data.

1) What is structured data?

- **Structured data** is generally tabular data that is represented by columns and rows in a database.
- Databases that hold tables in this form are called ***relational databases***.
- The mathematical term "*relation*" specify to a formed set of data held as a table.
- In structured data, all row in a table has the same set of columns.
- SQL (Structured Query Language) programming language used for structured data.

id	name	age
1	Jim	28
2	Pam	26
3	Michael	42

id	subject	Teacher
1	Languages	John Jones
2	Track	Wally West
3	Swimming	Arthur Curry
4	Computers	Victor Stone

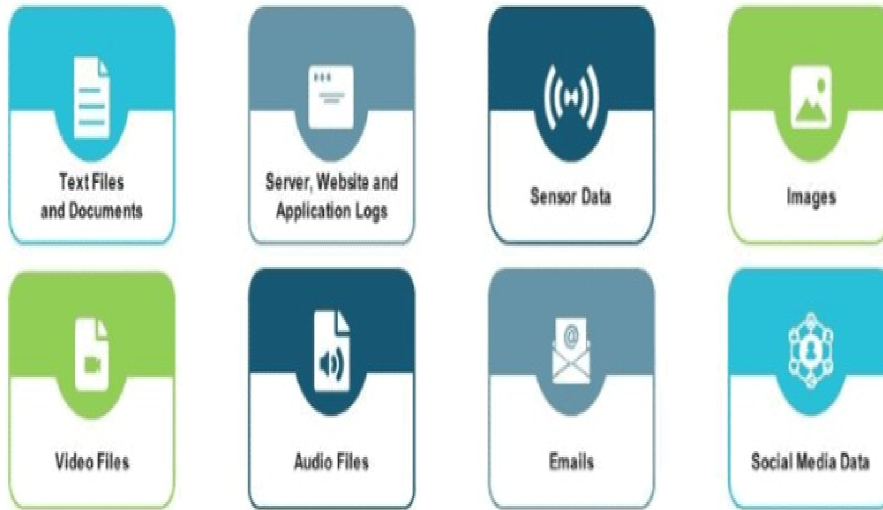
student_id	subject_id	grade
2	1	98
1	2	100
1	4	75
3	3	60
2	4	76
3	2	88

2) What is Semi structured Data

- **Semi-structured** data is information that doesn't consist of Structured data (relational database) but still has some structure to it.
- Semi-structured data consist of documents held in ***JavaScript Object Notation*** **(JSON) format**. It also includes ***key-value*** stores and ***graph*** databases.

```
## Document 1 ##
{
  "customerID": "103248",
  "name":
  {
    "first": "AAA",
    "last": "BBB"
  },
  "address":
  {
    "street": "Main Street",
    "number": "101",
    "city": "Acity",
    "state": "NY"
  },
  "ccOnFile": "yes",
  "firstOrder": "02/28/2003"
}
```

- **Unstructured data** is information that either does not organize in a pre-defined manner or not have a pre-defined data model.
- Unstructured information is a set of text-heavy but may contain data such as numbers, dates, and facts as well.
- **Videos, audio, and binary** data files might not have a specific structure. They're assigned to as **unstructured** data.



Structured Data vs Unstructured Data vs Semi-Structured:

Structured data is stored is predefined format and is highly specific; whereas **unstructured data** is a collection of many varied data types which are stored in their native formats; while **semi structured data** that does not follow the tabular data structure models associated with relational databases or other data table forms.

Pros and Cons of Structured Data

Pros	Cons
Requires less processing in comparison to unstructured data and is easier to manage.	Limited usability because of its pre-defined structure/format
Machine algorithms can easily crawl and use structured data which simplifies querying	Structured data is stored in data warehouses which are built for space saving but are difficult to change and not very scalable/flexible.
As an older format of data, there are several tools available for structured data that simplify usage, management, and analysis	

Pros and Cons of Unstructured Data

Pros	Cons

Variety of native formats facilitate a greater number of use-cases and applications	makes it equally challenging to analyze and leverage unstructured data.
As there is no need to predefine data, unstructured data is collected quickly and easily.	The large volume and undefined formats make data management a challenge and specialized tools a necessity.
Unstructured data is stored in on-premises or cloud data lakes which are highly scalable.	
Although challenging, the greater volume of unstructured data provides better insights and more opportunities to turn your data into a competitive advantage.	

Use cases for Structured data

Examples of structured data include names, dates, addresses, credit card numbers, stock information,geolocation,andmore.
Structured data is highly organized and easily understood by machine language. Those working within relational databases can input, search, and manipulate structured data relatively quickly using a relational database management system (RDBMS).

Use cases for Semi-Structured data

The use of semi-structured data enables us to integrate data from various sources or exchange data between different systems. Applications and systems need to evolve with time, but if we work purely with structured data, this is not possible. Let's consider web forms. You may want to modify forms and capture different data for different users. If you are using a traditional relational database, the database schema needs to be changed each time a new field is needed, and fields can not be left empty. Semi-structured data can allow you to capture any data in any structure without making changes to the database schema or coding. Adding or removing data does not impact functionality or dependencies.

Use cases for unstructured data

Here are a few examples where unstructured data is being used in analytics today.

Classifying image and sound. Using deep learning, a system can be trained to recognize images and sounds. The systems learn from labeled examples in order to accurately classify new images or sounds.
As input to predictive models. Text analytics — using natural language processing (NLP) or machine learning — is being used to structure unstructured text.
Chatbots in customer experience. Chatbots have been in the market for a number of years, but the newer ones have a better understanding of language and are more interactive.

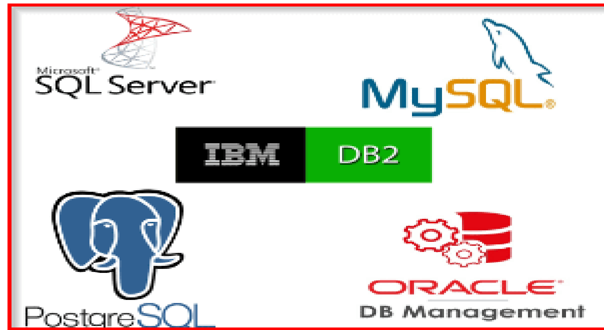
Characteristics Of Structured (Relational) and Unstructured (Non-Relational) Data

Relational Data

- Relational databases provide undoubtedly the most well-understood model for holding data.

inflexible structure can cause some problems.

- We can communicate with relational databases using Structured Query Language (**SQL).**
- SQL allows the joining of tables using a few lines of code, with a structure most beginner employees can learn very fast.
- Examples of relational databases:
  - MySQL
  - PostgreSQL
  - Db2

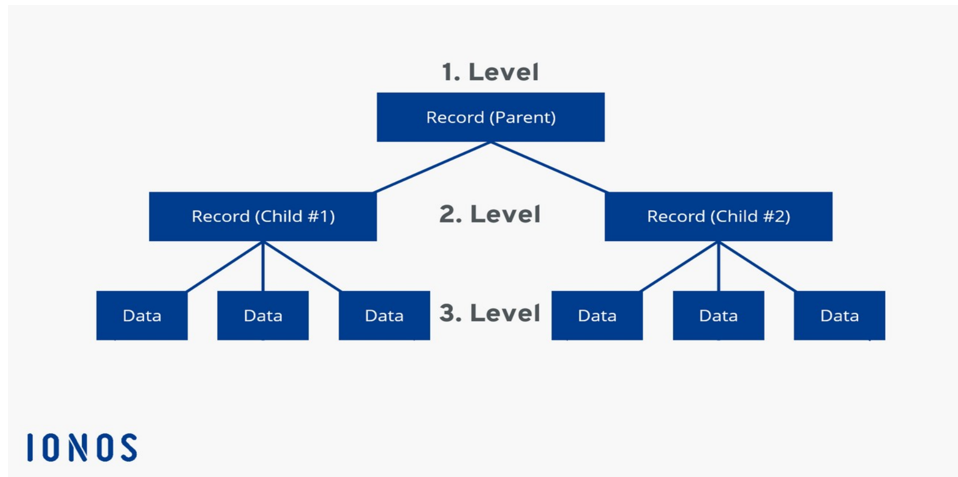


Non-Relational Data

- Non-relational databases permit us to store data in a format that more closely meets the original structure.
- A ***non-relational database*** is a database that does not use the tabular schema of columns and rows found in most traditional database systems.
- It uses a storage model that is enhanced for the specific requirements of the type of data being stored.
- In a non-relational database the data may be stored as **JSON documents**, as simple **key/value pairs**, or as a graph consisting of edges and vertices.
- Examples of non-relational databases:
  - Redis
  - JanusGraph
  - MongoDB
  - RabbitMQ

Structured Data words on the basis of relational database tables. Semi-Structured Data works on the basis of Relational Data Framework (RDF) or XML. Unstructured data works on the basis of binary data and the available characters. The data depends a lot on the schema.

XML data is hierarchical; **relational data** is represented in a model of logical relationships An XML document contains information about the relationship of data items to each other in the form of the hierarchy. With the relational model, the only types of relationships that can be defined are parent table and dependent table relationships.



A **hierarchical database model** is a data model in which the data are organized into a tree-like structure. The data are stored as **records** which are connected to one another through **links**. A record is a collection of fields, with each field containing only one value. The **type** of a record defines which fields the record contains.

The hierarchical database model mandates that each child record has only one parent, whereas each parent record can have one or more child records. In order to retrieve data from a hierarchical database, the whole tree needs to be traversed starting from the root node. This model is recognized as the first database model created by IBM in the 1960s.

Examples of hierarchical data represented as relational tables[edit]

An organization could store employee information in a table that contains attributes/columns such as employee number, first name, last name, and department number. The organization provides each employee with computer hardware as needed, but computer equipment may only be used by the employee to which it is assigned. The organization could store the computer hardware information in a separate table that includes each part's serial number, type, and the employee that uses it. The tables might look like this:

employee table

EmpNo	First Name	Last Name	Dept. Num
100	Almukhtar	Khan	10-L
101	Gaurav	Soni	10-L
102	Siddhartha	Soni	20-B
103	Siddhant	Soni	20-B

computer table

Serial Num	Type	User EmpNo
3009734-4	Computer	100
3-23-283742	Monitor	100
2-22-723423	Monitor	100

In this model, the `employee` data table represents the "parent" part of the hierarchy, while the `computer` table represents the "child" part of the hierarchy. In contrast to tree structures usually found in computer software algorithms, in this model the children point to the parents. As shown, each employee may possess several pieces of computer equipment, but each individual piece of computer equipment may have only one employee owner.

Consider the following structure:

EmpNo	Designation	ReportsTo
10	Director	
20	Senior Manager	10
30	Typist	20
40	Programmer	20

In this, the "child" is the same type as the "parent". The hierarchy stating EmpNo 10 is boss of 20, and 30 and 40 each report to 20 is represented by the "ReportsTo" column. In Relational database terms, the ReportsTo column is a foreign key referencing the EmpNo column. If the "child" data type were different, it would be in a different table, but there would still be a foreign key referencing the EmpNo column of the employees table.

XML DOCUMENT STRUCTURE

An **XML (EXtensible Markup Language) Document** contains declarations, elements, text, and attributes. It is made up of entities (storing units) and It tells us the structure of the data it refers to. It is used to provide a standard format of data transmission. As it helps in message delivery, it is not always stored physically, i.e. in a disk but generated dynamically but its structure always remains the same.

XML STANDARD STRUCTURE AND ITS RULES:

Rule 1: Its standard format consists of an **XML prolog** which contains both XML Declaration and XML DTD (Document Type Definition) and the body. If the XML prolog is present, it should always be the beginning of the document. The XML Version, by default, is **1.0**, and including only this forms the shortest XML Declaration. **UTF-8** is the default character encoding and is one of seven character-encoding schemes. If it is not present, it can result in some encoding errors.

Syntax of XML Declaration:

<?xml version="1.0" encoding="UTF-8"?>

Syntax of DTD:

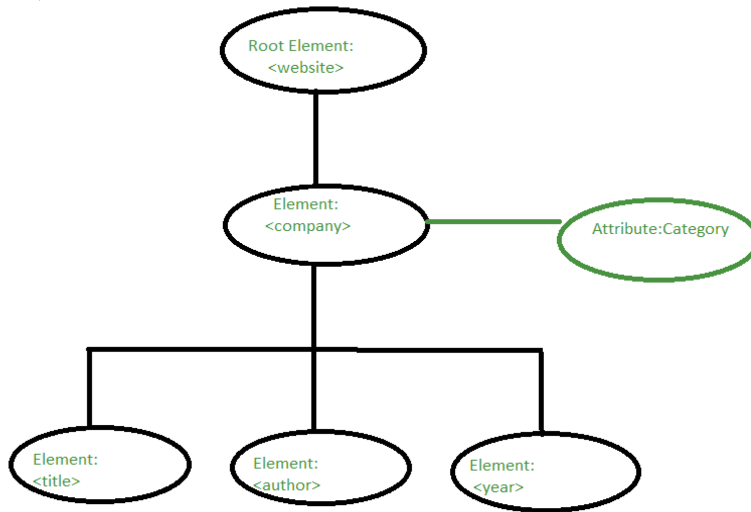
<!DOCTYPE root-element [<!element-declarations>]>

Example:

- XML

```
<!DOCTYPE website [
 <!ELEMENT website (name,company,phone)>
 <!ELEMENT name (#PCDATA)>
 <!ELEMENT company (#PCDATA)>
 <!ELEMENT phone (#PCDATA)>
]>
 <website>
 <name>GeeksforGeeks</name>
 <company>GeeksforGeeks</company>
 <phone>011-24567981</phone>
</website>
```

Rule 2: XML Documents must have a root element (the supreme parent element) and its child elements (sub-elements). To have a better view of the hierarchy of the data elements, XML follows the XML tree structure which comprises of one single root (parent) and multiple leaves (children).



Source Code of the above diagram:

- XML

```
<?xml version="1.0" encoding="UTF-8"?>
<website>
  <company category="geeksforgeeks">
    <title>Machine learning</title>
    <author>aarti majumdar</author>
    <year>2022</year>
  </company>
  <company category="geeksforgeeks">
    <title>Web Development</title>
    <author>aarti majumdar</author>
    <year>2022</year>
  </company>
```

```
    <company category="geeksforgeeks">
      <title>XML</title>
      <author>aarti majumdar</author>
      <year>2022</year>
    </company>
</website>
```

Rule 3: All XML Elements are required to have Closing and opening Tags(similar to HTML).
<message>Welcome to GeeksforGeeks</message>
Rule 4: The opening and closing tags are case-sensitive.
For Example, **<Message>** is different from **<message>** from above example.
Rule 5: Values of XML attributes are required to have quotations:

- XML

```
<website category="open source">
   <company>geeksforgeeks</company>
</website>
```

Rule 6: White-Spaces are retained and maintained in XML.

- XML

```
<message>welcome    to          geeksforgeeks</message>
```

Rule 7: Comments can be defined in XML enclosed between <!– and –> tags.

- XML

```
<!-- XML Comments are defined like this -->
```

Rule 8: XML elements must be nested properly.

- XML

```
<message>
   <company>GeeksforGeeks</company>
</message>
```

XML document is a well-organized collection of components and associated markup. An XML document can hold a wide range of information. For instance, a database having numbers or a mathematical equation etc.
Example: A simple document can be created as –

```
<?xml version = "1.0"?>

<Student-info>

  <name>Tanya Bajaj</name>

  <Organization>GeeksForGeeks</Organization>

  <contactNumber>(+91)-9966778909</contactNumber>
```

XML document has 2 sections:

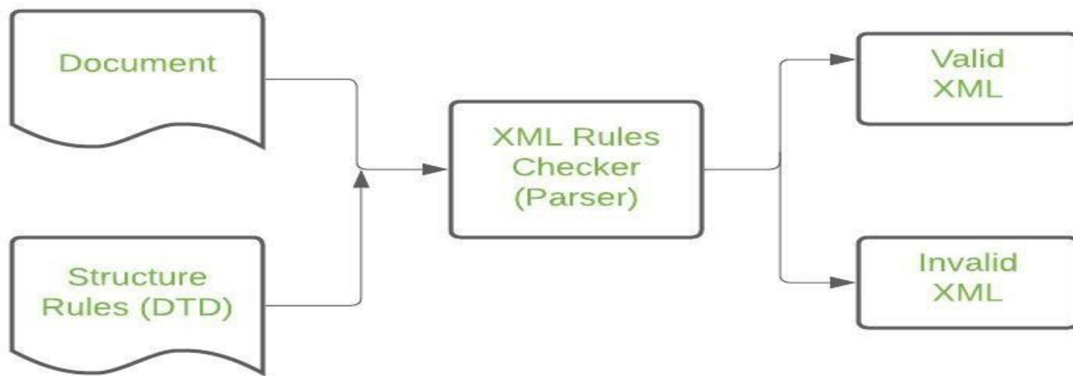
- **Document Prolog:** It contains XML & document type declaration (First 2 lines in the above XML doc).
- **Document Elements:** The building components of XML are Document Elements. These sections divide the document into a hierarchy of sections, each with its own utility.

Can the structure of the XML Document be checked somehow?

A tool for creating rules that regulate how documents are constructed is included in XML. These are referred to as DTDs (Document Type Definitions) in jargon. You may set up a DTD to check XML documents automatically in a variety of ways. Here is a couple of such examples:

- An optical title, a given name, and a surname make up a person's name.
- One or more channels can be found on a television schedule. There are one or more time slots on each channel. There is a program title and an optional description for each time slot.

These effects can be achieved by identifying the element types that you want to employ in your document and indicating the structural order in which they can appear in the document type. A utility called an XML Parser is then able to test whether or not the document meets the prescribed rules.



Checking the structure of XML Doc using XML Parser

An XML parser is a software library or package that provides client applications with interfaces for working with XML documents. In other words, The XML parser is a program that reads XML and allows a program to use it. The XML parser checks the document and ensures that it is properly formatted. XML parsers are included in most modern browsers.

What is an XML file?

An XML file contains XML code and ends with the file extension ".xml". It contains tags that define not only how the document should be structured but also how it should be stored and transported over the internet.

Let's look at a basic example of an XML file below. You can also click here to view the file directly in your browser.

```xml
    <student id="1">
        <firstName>Greg</firstName>
        <lastName>Dean</lastName>
        <certificate>True</certificate>
        <scores>
            <module1>70</module1>
            <module12>80</module12>
            <module3>90</module3>
        </scores>
    </student>
    <student ind="2">
        <firstName>Wirt</firstName>
        <lastName>Wood</lastName>
        <certificate>True</certificate>
        <scores>
            <module1>80</module1>
            <module12>80.2</module12>
            <module3>80</module3>
        </scores>
    </student>
</studentsList>
```

Image Source

As you can see, this file consists of plain text and tags. The plain text is shown in black and the tags are shown in green.

Plain text is the actual data being stored. In this example, the XML is storing student names as well as test scores associated with each student.

While plain text represents the data, tags indicate what the data is. Each tag represents a type of data, like "first name," "last name," or "score," and tells the computer what to do with the plain text data inside of it. Tags aren't supposed to be seen by users, only the software itself.

XML Hierarchy

Each instance of an XML tag is called an element. In an XML file, elements are arranged in a hierarchy, which means that elements can contain other elements.

The topmost element is called the "root" element, and contains all other elements, which are called "child" elements.

In the example above, "studentsList" is the root element. It contains two "student" elements. Each "student" element contains the elements "firstName," "lastName," "scores," etc. The beginning and end of each element are represented by a starting tag (e.g., "<firstName>") and a closing tag (e.g., "</firstName>") respectively.

in our example. This makes the file easier for humans to read, and does not affect how computers process the code.

XML Language

XML, short for "eXtensible Markup Language," was published by the World Wide Web Consortium (W3C) in 1998 to meet the challenges of large-scale electronic publishing. Since then, it has become one of the most widely used formats for sharing structured information among people, computers, and networks.

Since XML can be read and interpreted by people as well as computer software, it is known as human- and machine-readable.

The primary purpose of XML, however, is to store data in a way that can be easily read by and shared between software applications. Since its format is standardized, XML can be shared across systems or platforms, both locally and over the internet, and the recipient will still be able to parse the data.

It's important to understand that XML doesn't do anything with the data other than store it, like a database. Another piece of software must be created or used to send, receive, store, or display the data.

At this point, you might be thinking XML sounds a lot like another markup language, the Hypertext Markup Language (HTML). Let's take a closer look at the differences between these languages below.

Besides their purpose, there's one other key difference between XML and HTML tags.

When programming in HTML, a developer must use tags from the HTML tag library, or a standardized set of tags. While you can do a lot with these tags, there is a limited number available. That means there are only so many ways you can structure content on a web page.

XML does not have this limitation, as there is no preset library of XML tags. Instead, developers can create an unlimited number of custom tags to fit their data needs. This extensive customization is the "X" in XML.

To create custom tags, a developer writes a Document Type Definition (DTD), which is XML's version of a tag library. An XML file's DTD is indicated at the top of the file, and tells the software what each tag means and what to do with it.

For instance, an XML file containing info for a reservation system might have a custom "<res_start>" tag to define a time when a reservation begins. By reading the DTD, a program processing this file will know what the code "<res_start>7:00 PM PST</res_start>" means, and can use the information within the tag accordingly. This could mean sending this data in a confirmation email or storing it in another database.

stored in XML files can be read by computer programs with the help of custom tags, which indicate the type of element.

What is an XML file used for?

Since XML files are plain text documents, they are easy to create, store, transport, and interpret by computers and humans alike. This is why XML is one of the most commonly used languages on the internet. Many web-based software applications store information and send information to other apps in XML format.

Here are the most common uses of XML today:

Transporting Digital Information

The text-based format of XML files makes them highly portable, and therefore widely used for transferring information between web servers. Certain APIs, namely SOAP APIs and REST APIs, send information to other applications packaged in XML files.

Web Searching

Since XML defines the type of information contained in a document, it's easier and more effective to search the web with than HTML, for example.

Let's say you want to search for songs by Taylor Swift. Using HTML, you'd likely get back search results including interviews and articles that mention her songs. Using XML, search results would be restricted to songs only.

Computer Applications

XML files allow computer apps to easily structure and fetch the data that they need. After retrieving data from the file, programs can decide what to do with the data. This could mean storing in another database, using it in the program backend, or displaying it on the screen.

Additionally, some popular file formats are built with XML. Consider the Microsoft Office file extensions .docx (for Word documents), .xlsx (for Excel spreadsheets), and .pptx (for PowerPoint presentations). The "x" at the end of these file extensions stands for XML.

Websites and Web Apps

Websites and web apps can pull content for their pages from XML files. This is a common example of how the markup languages XML and HTML work together.
XML code modules might even appear within an HTML file in order to help display content on the page. This makes XML especially applicable to interactive websites and pages whose content changes dynamically. Depending on the user or screen size, an HTML file can choose to display only certain elements in the XML code, providing visitors with a personalized browsing experience.

How to Open an XML File

Since XML files are text files, you can open them in a few different ways. If you’re occasionally viewing XML files, you can open them directly in your favorite browser. If you’re frequently

computer.

In this section, I'll cover how to open XML files with each of these programs.

How to Open XML Files With a Web Browser

All modern web browsers allow you to read XML files right in the browser window. Like with the menu example from earlier, you can select an XML file from your device and choose to open it with your web browser. Here's how a file looks in Google Chrome:

While the appearance of the text will differ by browser, you should be able to easily parse the contents of the file, and you might also be able to hide and reveal specific elements.

If there's an error in the file, your browser will tell you with an error window. Google Chrome will display an error message like the following:

Note that your browser won't let you edit the file this way. To change the file, you'll need to use a specialized tool.

How to Open XML Files With an Online XML Editor

You can use a free online text file editor to view your XML files, change their contents, or convert them to other file formats. We recommend Code Beautify's XML Viewer for this purpose.
In the tool, click **Browse** to upload a file from your computer. Once uploaded, you can edit the file on the left and view the hierarchy of the XML contents on the right.

Once finished editing, click **Save & Share** to create a fresh XML file.
Code Beautify also offers many free conversion tools to convert your XML files to other popular data storage formats like JSON and CSV.

How to Open XML Files With a Text Editor

As with any text file, you can open XML files in any text editor. However, common editors like Notepad and Word probably won't display your XML files with colors or indentation. This makes the files less readable, as seen in the example below.

```
3         <firstName>Greg</firstName>
4         <lastName>Dean</lastName>
5         <certificate>True</certificate>
6         <scores>
7             <module1>70</module1>
8             <module12>80</module12>
9             <module3>90</module3>
10        </scores>
11    </student>
12    <student ind="2">
13        <firstName>Wirt</firstName>
14        <lastName>Wood</lastName>
15        <certificate>True</certificate>
16        <scores>
17            <module1>80</module1>
18            <module12>80.2</module12>
19            <module3>80</module3>
20        </scores>
21    </student>
22 </studentsList>
```

You'll want to opt for a specialized text editor that will detect the .xml format and display your files accordingly. For PCs, Notepad++ is a popular option. For Macs, try Xmplify or Eclipse. Alternatively, you can use a simple text editor and apply indentation to your files with a free online XML formatter.

If any of your systems implement XML files, they will almost certainly write all of these files for you. If you want to practice writing your own basic XML files, you can do so in a text editor. Let's walk through how to create an XML file below.

How to Create an XML File

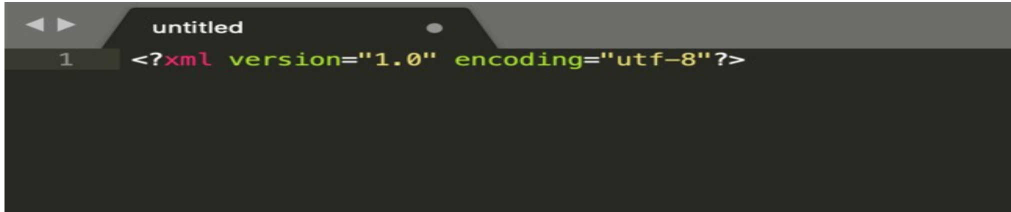
1. Open your text editor of choice.
2. On the first line, write an XML declaration.
3. Set your root element below the declaration.
4. Add your child elements within the root element.
5. Review your file for errors.
6. Save your file with the .xml file extension.
7. Test your file by opening it in the browser window.

1. Open your text editor of choice.

I'll use Sublime Text for this demo since it's free and works on macOs, Linux, and Microsoft operating systems.

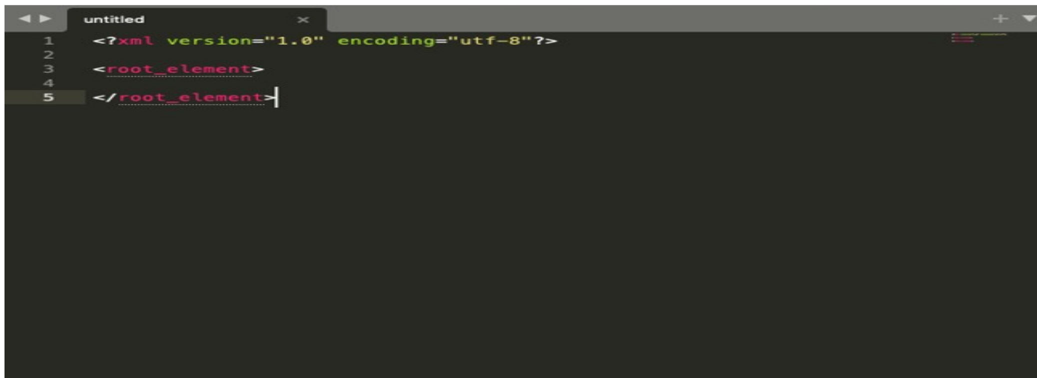
2. On the first line, write an XML declaration.

This declaration tells the application running the file that the language is XML.



3. Set your root element below the declaration.

Every XML file has one root element, which contains all other child elements. The root element is written below the declaration.



In this example file, "<root_element>" is the starting tag for the root element, and "</root_element>" is the closing element. All other elements will go between these tags.

You can substitute "root_element" in both tags with a name relevant to the information you're storing.

4. Add your child elements within the root element.

Next, add your child elements between the starting and closing tag of the root element. You can nest a child element within another child element.

Like the root element, each child element needs a starting tag and a closing tag. After adding child tags, your file will look something like this:

```
<?xml version="1.0" encoding="utf-8"?>

<root_element>

    <child_element_1>
        <child_element_2>Content</child_element_2>
    </child_element_1>

    <child_element_1>
        <child_element_2>Content</child_element_2>
    </child_element_1>

    <child_element_1>
        <child_element_2>Content</child_element_2>
        <child_element_2>Content</child_element_2>
    </child_element_1>

</root_element>
```

Instances of "root_element", "child_element", and "Content" can be swapped with names that make more sense for your file.

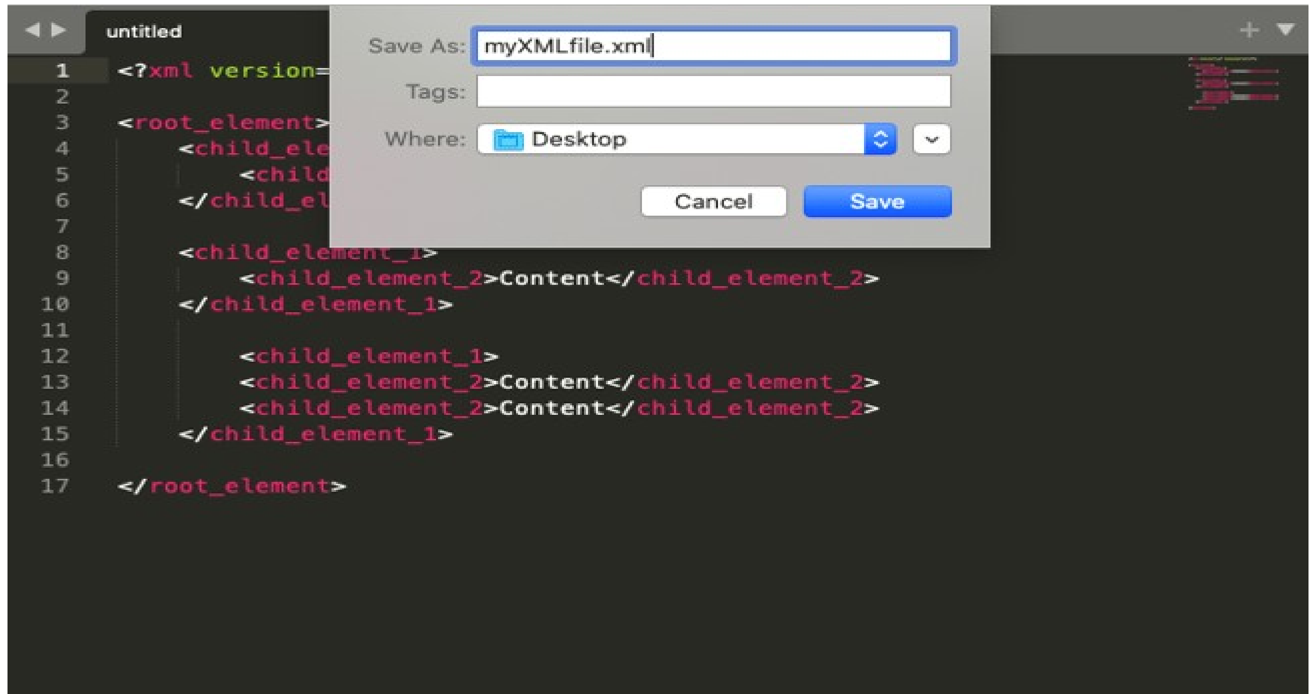
5. Review your file for errors.

Time to review. Are there any missing closing tags? Any rogue ampersands? Does the document type declaration appear after the first element in the document? These are just a few possible errors.

Notice that line 5 is highlighted below. That's because the closing tag of the "child_element_2" is missing a bracket.

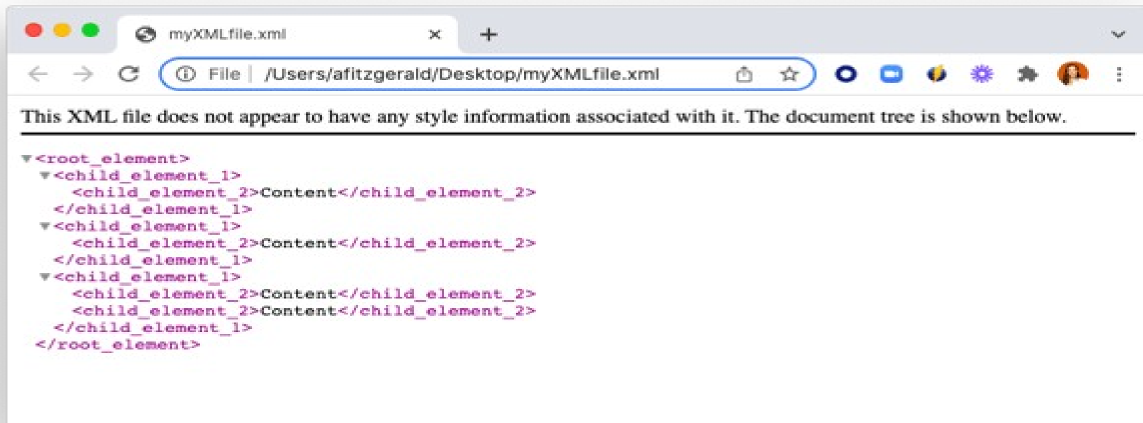
```
untitled

<?xml version="1.0" encoding="utf-8"?>

<root_element>
    <child_element_1>
        <child_element_2>Content</child_element_2
    </child_element_1>

    <child_element_1>
        <child_element_2>Content</child_element_2>
    </child_element_1>

        <child_element_1>
        <child_element_2>Content</child_element_2>
        <child_element_2>Content</child_element_2>
    </child_element_1>

</root_element>
```

As said above, an XML file ends with the file extension ".xml". So make sure to save your file with that extension.



7. Test your file by opening it in the browser window.

Finally, test that your file is working by dragging and dropping it into a new browser tab or window.



XML SCHEMA

What is an XML Schema?
An XML Schema describes the structure of an XML document.
The XML Schema language is also referred to as XML Schema Definition (XSD).

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

The purpose of an XML Schema is to define the legal building blocks of an XML document:

- the elements and attributes that can appear in a document
- the number of (and order of) child elements
- data types for elements and attributes
- default and fixed values for elements and attributes

Why Learn XML Schema?

In the XML world, hundreds of standardized XML formats are in daily use.
Many of these XML standards are defined by XML Schemas.
XML Schema is an XML-based (and more powerful) alternative to DTD.

XML Schemas Support Data Types

One of the greatest strength of XML Schemas is the support for data types.

- It is easier to describe allowable document content
- It is easier to validate the correctness of data
- It is easier to define data facets (restrictions on data)
- It is easier to define data patterns (data formats)
- It is easier to convert data between different data types

XML Schemas use XML Syntax

Another great strength about XML Schemas is that they are written in XML.

- You don't have to learn a new language
- You can use your XML editor to edit your Schema files
- You can use your XML parser to parse your Schema files
- You can manipulate your Schema with the XML DOM
- You can transform your Schema with XSLT

XML Schemas are extensible, because they are written in XML.
With an extensible Schema definition you can:

- Reuse your Schema in other Schemas
- Create your own data types derived from the standard types
- Reference multiple schemas in the same document

XML Schemas Secure Data Communication

When sending data from a sender to a receiver, it is essential that both parts have the same "expectations" about the content.

understand.
A date like: "03-11-2004" will, in some countries, be interpreted as 3.November and in other countries as 11.March.
However, an XML element with a data type like this:
<date type="date">2004-03-11</date>
ensures a mutual understanding of the content, because the XML data type "date" requires the format "YYYY-MM-DD".

A well-formed XML document is a document that conforms to the XML syntax rules, like:

- it must begin with the XML declaration
- it must have one unique root element
- start-tags must have matching end-tags
- elements are case sensitive
- all elements must be closed
- all elements must be properly nested
- all attribute values must be quoted
- entities must be used for special characters

Even if documents are well-formed they can still contain errors, and those errors can have serious consequences.

XML documents can have a reference to a DTD or to an XML Schema.

A Simple XML Document

Look at this simple XML document called "note.xml":

```
<?xml version="1.0"?>
<note>
 <to>Tove</to>
 <from>Jani</from>
 <heading>Reminder</heading>
 <body>Don't forget me this weekend!</body>
</note>
```

A DTD File

The following example is a DTD file called "note.dtd" that defines the elements of the XML document above ("note.xml"):

```
<!ELEMENT note (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

Line 2-5 defines the to, from, heading, body elements to be of type "#PCDATA".

An XML Schema

The following example is an XML Schema file called "note.xsd" that defines the elements of the XML document above ("note.xml"):

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="https://www.w3schools.com"
xmlns="https://www.w3schools.com"
elementFormDefault="qualified">
<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

The note element is a **complex type** because it contains other elements. The other elements (to, from, heading, body) are **simple types** because they do not contain other elements.

XQUERY EXAMPLE

The XML Example Document
We will use the following XML document in the examples below.
"books.xml":

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
<book category="COOKING">
  <title lang="en">Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
  <price>30.00</price>
</book>
<book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
```

```
  <price>29.99</price>
</book>
<book category="WEB">
  <title lang="en">XQuery Kick Start</title>
  <author>James McGovern</author>
  <author>Per Bothner</author>
  <author>Kurt Cagle</author>
  <author>James Linn</author>
  <author>Vaidyanathan Nagarajan</author>
  <year>2003</year>
  <price>49.99</price>
</book>
<book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
</book>
</bookstore>
```

View the "books.xml" file in your browser.

How to Select Nodes From "books.xml"?

Functions

XQuery uses functions to extract data from XML documents.
The doc() function is used to open the "books.xml" file:

```
doc("books.xml")
```

Path Expressions

XQuery uses path expressions to navigate through elements in an XML document.
The following path expression is used to select all the title elements in the "books.xml" file:

```
doc("books.xml")/bookstore/book/title
```

(/bookstore selects the bookstore element, /book selects all the book elements under the bookstore element, and /title selects all the title elements under each book element)
The XQuery above will extract the following:

```
<title lang="en">Everyday Italian</title>
<title lang="en">Harry Potter</title>
<title lang="en">XQuery Kick Start</title>
<title lang="en">Learning XML</title>
```

Predicates

XQuery uses predicates to limit the extracted data from XML documents.
The following predicate is used to select all the book elements under the bookstore element that have a price element with a value that is less than 30:

```
doc("books.xml")/bookstore/book[price<30]
```

The XQuery above will extract the following:

```
<book category="CHILDREN">
  <title lang="en">Harry Potter</title>
```

```
  <year>2005</year>
  <price>29.99</price>
</book>
```

XPATH SYNTAX:

XPath uses path expressions to select nodes or node-sets in an XML document. The node is selected by following a path or steps.

The XML Example Document

We will use the following XML document in the examples below.

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
<book>
  <title lang="en">Harry Potter</title>
  <price>29.99</price>
</book>
<book>
  <title lang="en">Learning XML</title>
  <price>39.95</price>
</book>
</bookstore>
```

Selecting Nodes

XPath uses path expressions to select nodes in an XML document. The node is selected by following a path or steps. The most useful path expressions are listed below:

Expression	Description
nodename	Selects all nodes with the name "*nodename*"
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes

In the table below we have listed some path expressions and the result of the expressions:

Path Expression	Result
bookstore	Selects all nodes with the name "bookstore"
/bookstore	Selects the root element bookstore **Note:** If the path starts with a slash (/) it always represents an absolute path to an element!
bookstore/book	Selects all book elements that are children of bookstore

bookstore//book	Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element
//@lang	Selects all attributes that are named lang

Predicates
Predicates are used to find a specific node or a node that contains a specific value.
Predicates are always embedded in square brackets.
In the table below we have listed some path expressions with predicates and the result of the expressions:

Path Expression	**Result**
/bookstore/book[1]	Selects the first book element that is the child of the bookstore element. **Note:** In IE 5,6,7,8,9 first node is[0], but according to W3C, it is [1]. To solve this problem in IE, set the SelectionLanguage to XPath: *In JavaScript:* *xml*.setProperty("SelectionLanguage","XPath");
/bookstore/book[last()]	Selects the last book element that is the child of the bookstore element
/bookstore/book[last()-1]	Selects the last but one book element that is the child of the bookstore element
/bookstore/book[position()<3]	Selects the first two book elements that are children of the bookstore element
//title[@lang]	Selects all the title elements that have an attribute named lang
//title[@lang='en']	Selects all the title elements that have a "lang" attribute with a value of "en"
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00
/bookstore/book[price>35.00]/title	Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00

UNIT V

INFORMATION RETRIEVAL AND WEB SEARCH

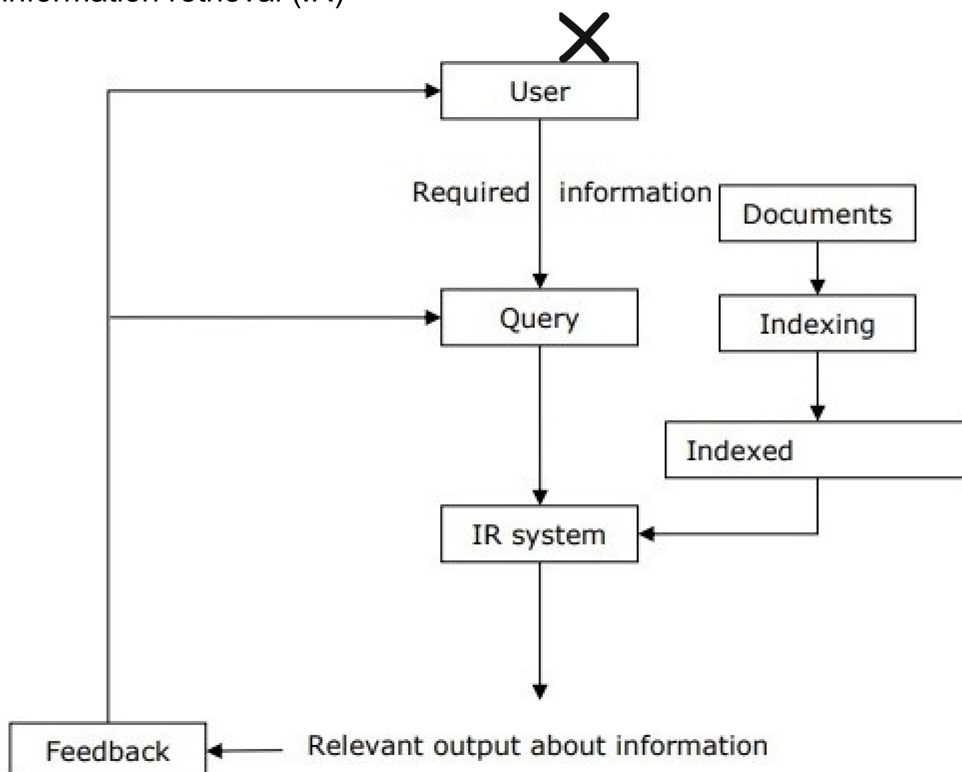
IR Concepts – Retrieval Models – Queries In Ir System – Text Preprocessing – Inverted Indexing – Evaluation Measures – Web Search And Analytics – Ontology Based Search - Current Trends.

IR CONCEPTS

An **information retrieval** (IR) system is a set of algorithms that facilitate the relevance of displayed documents to searched queries. In simple words, it works to sort and rank documents based on the queries of a user.

Information retrieval (IR) may be defined as a software program that deals with the organization, storage, retrieval and evaluation of information from document repositories particularly textual information. The system assists users in finding the information they require but it does not explicitly return the answers of the questions. It informs the existence and location of documents that might consist of the required information. The documents that satisfy user's requirement are called relevant documents. A perfect IR system will retrieve only relevant documents.

With the help of the following diagram, we can understand the process of information retrieval (IR) −



It is clear from the above diagram that a user who needs information will have to formulate a request in the form of query in natural language. Then the IR system will

respond by retrieving the relevant output, in the form of documents, about the required information.

RETRIEVAL MODELS

Information Retrieval (IR) Model

Mathematically, models are used in many scientific areas having objective to understand some phenomenon in the real world. A model of information retrieval predicts and explains what a user will find in relevance to the given query. IR model is basically a pattern that defines the above-mentioned aspects of retrieval procedure and consists of the following −

- A model for documents.
- A model for queries.
- A matching function that compares queries to documents.

Mathematically, a retrieval model consists of −

D − Representation for documents.

R − Representation for queries.

F − The modeling framework for D, Q along with relationship between them.

R (q,di) − A similarity function which orders the documents with respect to the query. It is also called ranking.

Types of Information Retrieval (IR) Model

An information model (IR) model can be classified into the following three models −

Classical IR Model

It is the simplest and easy to implement IR model. This model is based on mathematical knowledge that was easily recognized and understood as well. Boolean, Vector and Probabilistic are the three classical IR models.

Non-Classical IR Model

It is completely opposite to classical IR model. Such kind of IR models are based on principles other than similarity, probability, Boolean operations. Information logic model, situation theory model and interaction models are the examples of non-classical IR model.

Alternative IR Model

It is the enhancement of classical IR model making use of some specific techniques from some other fields. Cluster model, fuzzy model and latent semantic indexing (LSI) models are the example of alternative IR model.

QUERIES IN IR SYSTEM

An information retrieval (IR) query language is **a query language used to make queries into search index**. A query language is formally defined in a context-free grammar (CFG) and can be used by users in a textual, visual/UI or speech form.

The Information Retrieval (IR) system finds the relevant documents from a large data set according to the user query. Queries submitted by users to search engines might be ambiguous, concise and their meaning may change over time. Some of the types of Queries in IR systems are –

1. Keyword Queries :

- Simplest and most common queries.
- The user enters just keyword combinations to retrieve documents.
- These keywords are connected by logical AND operator.
- All retrieval models provide support for keyword queries.

2. Boolean Queries :

- Some IR systems allow using +, -, AND, OR, NOT, ( ), Boolean operators in combination of keyword formulations.
- No ranking is involved because a document either satisfies such a query or does not satisfy it.
- A document is retrieved for boolean query if it is logically true as exact match in document.

3. Phase Queries :

- When documents are represented using an inverted keyword index for searching, the relative order of items in document is lost.
- To perform exact phase retrieval, these phases are encoded in inverted index or implemented differently.
- This query consists of a sequence of words that make up a phase.
- It is generally enclosed within double quotes.

4. Proximity Queries :

- Proximity refers ti search that accounts for how close within a record multiple items should be to each other.
- Most commonly used proximity search option is a phase search that requires terms to be in exact order.
- Other proximity operators can specify how close terms should be to each other. Some will specify the order of search terms.
- Search engines use various operators names such as NEAR, ADJ (adjacent), or AFTER.
- However, providing support for complex proximity operators becomes expensive as it requires time-consuming pre-processing of documents and so it is suitable for smaller document collections rather than for web.

5. Wildcard Queries :

- It supports regular expressions and pattern matching-based searching in text.
- Retrieval models do not directly support for this query type.
- In IR systems, certain kinds of wildcard search support may be implemented. Example: usually words ending with trailing characters.

6. Natural Language Queries :

- There are only a few natural language search engines that aim to understand the structure and meaning of queries written in natural language text, generally as question or narrative.
- The system tries to formulate answers for these queries from retrieved results.
- Semantic models can provide support for this query type.

TEXT PREPROCESSING

Text preprocessing is **a method to clean the text data and make it ready to feed data to the model**. Text data contains noise in various forms like emotions, punctuation, text in a different case.

Tasks in data preprocessing

1. **Data Cleaning:** It is also known as scrubbing. This task involves filling of missing values, smoothing or removing noisy data and outliers along with resolving inconsistencies.
2. **Data Integration:** This task involves integrating data from multiple sources such as databases (relational and non-relational), data cubes, files, etc. The data sources can be homogeneous or heterogeneous. The data obtained from the sources can be structured, unstructured or semi-structured in format.
3. **Data Transformation:** This involves normalisation and aggregation of data according to the needs of the data set.
4. **Data Reduction:** During this step data is reduced. The number of records or the number of attributes or dimensions can be reduced. Reduction is performed by keeping in mind that reduced data should produce the same results as original data.

5. **Data Discretization:** It is considered as a part of data reduction. The numerical attributes are replaced with nominal ones.

Data Cleaning

The data cleaning process detects and removes the errors and inconsistencies present in the data and improves its quality. Data quality problems occur due to misspellings during data entry, missing values or any other invalid data. Basically, "dirty" data is transformed into clean data. "Dirty" data does not produce the accurate and good results. Garbage data gives garbage out. So it becomes very important to handle this data. Professionals spend a lot of their time on this step.

Reasons for "dirty" or "unclean" data

1. Dummy values
2. Absence of data
3. Violation of business rules
4. Data integration problems
5. Contradicting data
6. Inappropriate use of address line
7. Reused primary keys
8. Non-unique identifiers

What to do to clean data?

1. Handle Missing Values
2. Handle Noise and Outliers
3. Remove Unwanted data

Handle Missing Values

Missing values cannot be looked over in a data set. They must be handled. Also, a lot of models do not accept missing values. There are several techniques to handle missing data, choosing the right one is of utmost importance. The choice of technique to deal with missing data depends on the problem domain and the goal of data mining process. The different ways to handle missing data are:

1. **Ignore the data row:** This method is suggested for records where maximum amount of data is missing, rendering the record meaningless. This method is usually avoided where only less attribute values are missing. If all the rows with missing values are ignored i.e. removed, it will result in poor performance.
2. **Fill the missing values manually:** This is a very time consuming method and hence infeasible for almost all scenarios.
3. **Use a global constant to fill in for missing values:** A global constant like “NA” or 0 can be used to fill all the missing data. This method is used when missing values are difficult to be predicted.

4. **Use attribute mean or median:** Mean or median of the attribute is used to fill the missing value.
5. **Use forward fill or backward fill method:** In this, either the previous value or the next value is used to fill the missing value. A mean of the previous and succession values may also be used.
6. Use a data-mining algorithm to predict the most probable value

Data Integration

In this step, a coherent data source is prepared. This is done by collecting and integrating data from multiple sources like databases, legacy systems, flat files, data cubes etc.

Issues in Data Integration

1. **Schema Integration:** Metadata (i.e. the schema) from different sources may not be compatible. This leads to *entity identification problem*. Example : Consider two data sources R and S. Customer id in R is represented as cust_id and in S is represented is c_id. They mean the same thing, represent the same thing but have different names which leads to integration problems. Detecting and resolving them is very important to have a coherent data source.
2. **Data value conflicts:** The values or metrics or representations of the same data maybe different in for the same real world entity in different data sources. This leads to different representations of the same data, different scales

etc. Example : Weight in data source R is represented in kilograms and in source S is represented in grams. To resolve this, data representations should be made consistent and conversions should be performed accordingly.

3. **Redundant data:** Duplicate attributes or tuples may occur as a result of integrating data from various sources. This may also lead to inconsistencies. These redundancies or inconsistencies may be reduced by careful integration of data from multiple sources. This will help in improving the mining speed and quality. Also, co-relational analysis can be performed to detect redundant data.

Data Reduction

If the data is very large, data reduction is performed. Sometimes, it is also performed to find the most suitable subset of attributes from a large number of attributes. This is known as dimensionality reduction. Data reduction also involves reducing the number of attribute values and/or the number of tuples. Various data reduction techniques are:

1. **Data cube aggregation:** In this technique the data is reduced by applying OLAP operations like slice, dice or rollup. It uses the smallest level necessary to solve the problem.
2. **Dimensionality reduction:** The data attributes or dimensions are reduced. Not all attributes are required for data mining. The most suitable subset of attributes are

selected by using techniques like forward selection, backward elimination, decision tree induction or a combination of forward selection and backward elimination.

3. **Data compression:** In this technique. large volumes of data is compressed i.e. the number of bits used to store data is reduced. This can be done by using lossy or lossless compression. In *loss compression,* the quality of data is compromised for more compression. In *lossless compression,* the quality of data is not compromised for higher compression level.

4. **Numerosity reduction :** This technique reduces the volume of data by choosing smaller forms for data representation. Numerosity reduction can be done using histograms, clustering or sampling of data. Numerosity reduction is necessary as processing the entire data set is expensive and time consuming.

INVERTED INDEXING

An inverted index is an index data structure storing a mapping from content, such as words or numbers, to its locations in a document or a set of documents. In simple words, it is a hashmap like data structure that directs you from a word to a document or a web page.

There are two types of inverted indexes: A **record-level inverted index** contains a list of references to documents for each word. A **word-level inverted index** additionally contains the positions of each word within a document. The latter form offers more functionality, but needs more processing power and space to be created.
Suppose we want to search the texts “hello everyone, ” “this article is based on inverted index, ” “which is hashmap like data structure”. If we index by (text, word within the text), the index with location in text is:

```
hello                    (1, 1)
```

```
everyone          (1, 2)
this              (2, 1)
article           (2, 2)
is                (2, 3); (3, 2)
based             (2, 4)
on                (2, 5)
inverted          (2, 6)
index             (2, 7)
which             (3, 1)
hashmap           (3, 3)
like              (3, 4)
data              (3, 5)
structure         (3, 6)
```

The word “hello” is in document 1 (“hello everyone”) starting at word 1, so has an entry (1, 1) and word “is” is in document 2 and 3 at ‘3rd’ and ‘2nd’ positions respectively (here position is based on word).
The index may have weights, frequencies, or other indicators.

Steps to build an inverted index:

- **Fetch the Document**
  Removing of Stop Words: Stop words are most occurring and useless words in document like “I”, “the”, “we”, “is”, “an”.
- **Stemming of Root Word**
  Whenever I want to search for “cat”, I want to see a document that has information about it. But the word present in the document is called “cats” or “catty” instead of “cat”. To relate the both words, I’ll chop some part of each and every word I read so that I could get the “root word”. There are standard tools for performing this like “Porter’s Stemmer”.
- **Record Document IDs**
  If word is already present add reference of document to index else create new entry. Add additional information like frequency of word, location of word etc.

Example:

```
Words             Document
ant               doc1
demo              doc2
world             doc1, doc2
```

Advantage of Inverted Index are:

- Inverted index is to allow fast full text searches, at a cost of increased processing when a document is added to the database.
- It is easy to develop.
- It is the most popular data structure used in document retrieval systems, used on a large scale for example in search engines.

Inverted Index also has disadvantage:

- Large storage overhead and high maintenance costs on update, delete and insert.

WEB SEARCH AND ANALYTICS

Searching for information on the Web. The term may be used to differentiate Web searching from searching local users' PCs or servers in the company datacenter. See Web search engines.

What is Web Analytics?

Web Analytics is the methodological study of **online/offline** patterns and trends. It is a technique that you can employ to collect, measure, report, and analyze your website data. It is normally carried out to analyze the performance of a website and optimize its web usage.

We use web analytics to track key metrics and analyze visitors' activity and traffic flow. It is a tactical approach to collect data and generate reports.

Importance of Web Analytics

We need Web Analytics to assess the success rate of a website and its associated business. Using Web Analytics, we can −

- Assess web content problems so that they can be rectified
- Have a clear perspective of website trends
- Monitor web traffic and user flow
- Demonstrate goals acquisition
- Figure out potential keywords
- Identify segments for improvement
- Find out referring sources

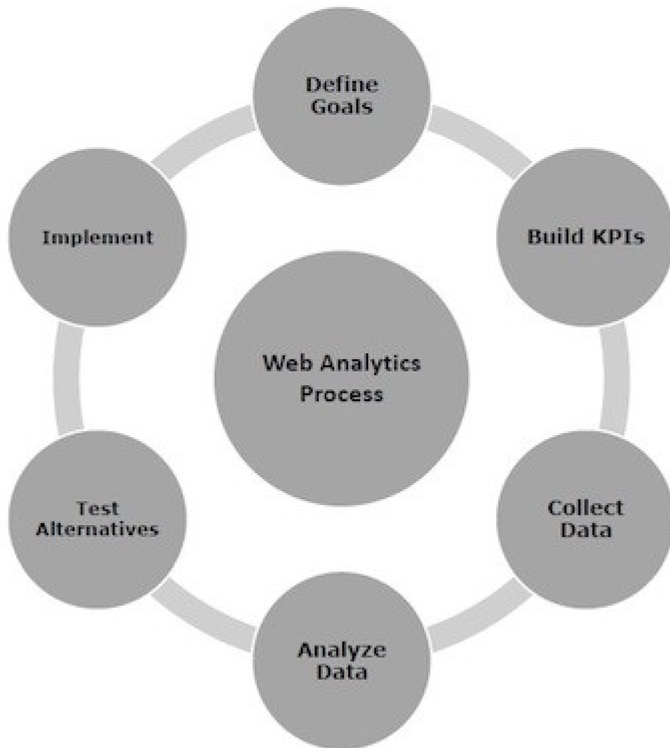
Web Analytics Process

The primary objective of carrying out Web Analytics is to optimize the website in order to provide better user experience. It provides a data-driven report to measure visitors' flow throughout the website.

Take a look at the following illustration. It depicts the process of web analytics.

- Set the business **goals**.

- To track the goal achievement, set the **Key Performance Indicators** (KPI).
- **Collect** correct and suitable data.
- To extract insights, **Analyze** data.
- Based on assumptions learned from the data analysis, **Test alternatives**.
- Based on either data analysis or website testing, **Implement insights**.



Web Analytics is an ongoing process that helps in attracting more traffic to a site and thereby, increasing the Return on Investment.

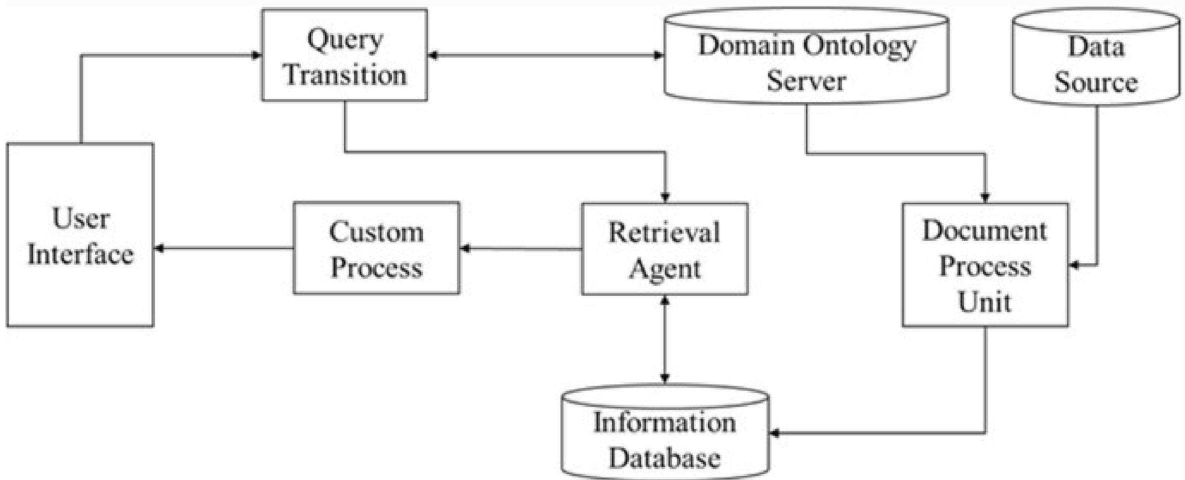
ONTOLOGY BASED SEARCH

Ontology-based data integration involves **the use of one or more ontologies to effectively combine data or information from multiple heterogeneous sources**. It is one of the multiple data integration approaches and may be classified as Global-As-View (GAV).

The concept in domain ontology has a relation to other concepts simultaneously. The interrelation between concepts of the semantic relative network implements synonym expansion retrieval, semantic entailment expansion, semantic correlation expansion. We introduce a domain ontology information retrieval model to apply ontology into the traditional information retrieval model by query expansion to improve efficiency.

An illustration of structure for the information retrieval model is shown in Fig. 1.

Fig. 1

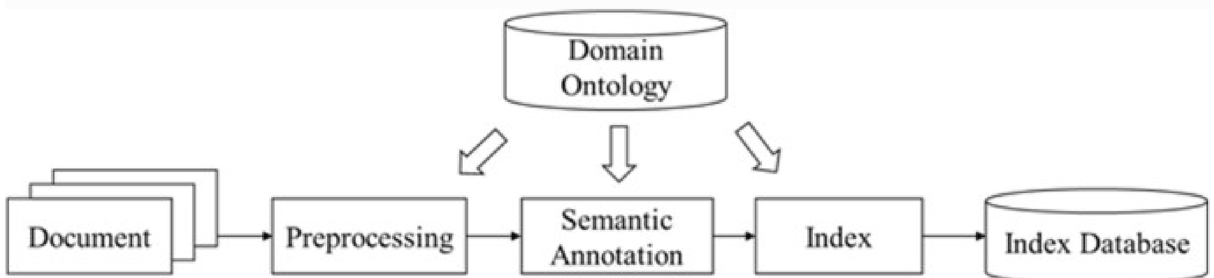


An illustration of structure for information retrieval model

The system consists of two parts: ontology document processing (including domain ontology servers, data source, document process unit and information database) and ontology document retrieval (including domain ontology server, query transition, custom process, and retrieval agent).

Ontology document processing

Document processing extracts useful information from an unstructured text message and establishes mapping relations between document terms and concepts based on domain ontology . The document processing is shown in Fig. 2.



In the preprocessing procedure, each document in the document set implements vocabulary, analyzes words, and filters numbers, hyphens, and punctuations. Using a stop word list removes function words to leave useful words such as noun and verb . Extracting stem words and removing the

prefix and postfix improve the accuracy of retrieval. Finally, determining certain words as an index element expresses literature content conception.

Annotating semantic on a retrieved object by analyzing characteristic vocabulary builds the mapping relation between words and concepts. First, characteristic words are extracted and the weight of each word is calculated by counting word frequency to distinguish the importance of words. In this paper, the genetic algorithm is used to calculate the best weighting factor. In the end, it is applied to the actual retrieval system.

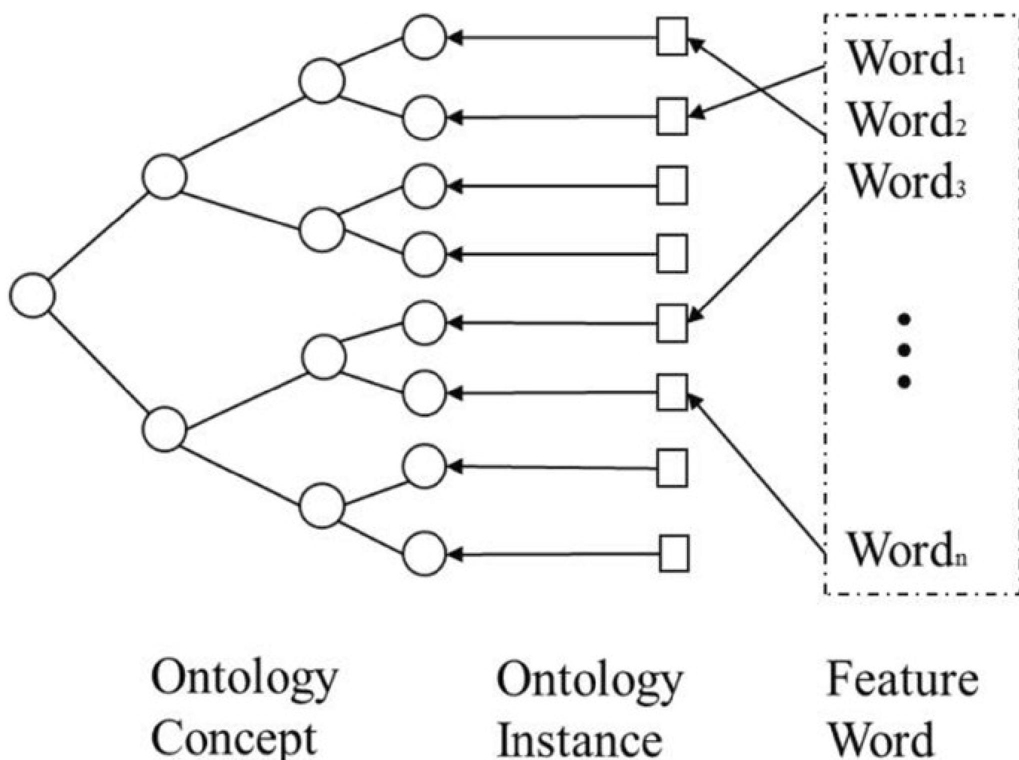
The system automatically learns weighted factor by genetic algorithm. It is a heuristic method which simulates biological evolution processes and through factor mutation eliminates the non-ideal factor sets and leaves the optimal factor set. The algorithm tries to maximize the fitness function as a parameter estimation to search a population consisting of the fittest individual; in our case, those are the parameters of weighted term in retrieving. In Fig. 3, the pseudo-code of genetic algorithm for weighted term frequency is described.

```
(1)  Input: paper list, expert rank list
(2)  Output: weight of factor: wtit, wkey, wabs
(3)  Initialize P(0)
(4)  for t=0:#max_iteration_times do
(5)      // Evaluate Fitness
(6)      P(t)=f(wtit, wkey, wabs) – f(wtit', wkey', wabs')
(7)      //Select operation to P(i)
(8)      Slice=random()*∑ P(t)
(9)      for i=1:#PopulationSize do
(10)         FitnessSoFar += P(i)
(11)if FitnessSoFar>slice
(12)             TheChosenOne = P(i)
(13)     end for
(14)     //Mutation operation to P(t)
(15)     for i=0:#ChromosomeSize do
(16)if random()<matitionRate
(17)         Chromo[i]+=(random()-0.5)*maxPerturbation
(18)if chrmo[i] > left
```

This algorithm simulates the evolution process by gradually adjusting weight factor and eliminating factor combination with a low fitness value. If the fitness result for one combination is lower than the other one, this group will be likely excluded in the next generation. To avoid the local optimization, we select many original generations and decrease the unqualified group time by time. In each iteration, the factor interval lies in $[w_i - 0.2, w_i + 0.4]$ to lower the negative factors. Fitness function *P(t)* determines how fit an individual is with new weighted combination $(w'_{tit}, w'_{key}, w'_{abs})$. The traditional factor set is replaced by *P(t)* with higher fitness, then calculated with a query word for similarity of each papers and generated the rank list. The penalty function *f* is used to get the distance of the expert list.

Then, for each semantic meaning of ontology term, whether it exists in extracting characteristic vocabulary is checked. If the semantic exists, the document and weight with semantic term is calculated to manifest the text with semantic information.

After document feature extraction, document index based on the concept to reflect the internal relation between text index terms is established and ambiguity during annotation is excluded. An index based on the concept consists of feature words with their relation given by semantic parsing. Feature words connect through ontology instance and documents. The structure of the ontology concept index is shown in Fig. 4.



Ontology document retrieval

The procedure of document retrieval is listed below:

1. 1)

   The user inputs search words or phrases in the search interface, then the system removes function words and reserves noun and verb. Term extraction from words is implement to get semantic conceptual words and phrases. The result is passed to the query transition module.

2. 2)

   The query transition module sends consequence to the ontology server to search for a corresponding semantic concept, including hypernym, hyponym, synonym, and conceptual meanings [17]. If the word is not found in the ontology database, it returns to the user to help in adjusting the retrieval strategy.

3. 3)

   For the matching concept in domain ontology, the query transition module implements search, semantic judgment, and query extension to add semantic information to query. The module submits query to a retrieval agent for searching. For words with an uncertain semantic message, execute a keyword matching method to search.

4. 4)

   Handled by the custom process module, the user interface list query results according to exact word, synonym, hypernym, and hyponym words.

Before the retrieval process, the system executes semantic analysis for the user query request. Keyword is extracted from stop words and whether keyword belongs to ontology database is checked. Through combining concepts in ontology library, more semantic information is obtained by semantic reasoning. The pseudo-code of query semantic analysis algorithm is shown in Fig. 5.

Fig. 5

```
(1)  Input: query expression
(2)  Output: semantic concept set TermsWithOntologyConcept
(3)  Stem the query expression.
(4)  Get terms (q1, q2, ..., qn) using stop list.
(5)  Count terms weights (w1, w2, ..., wn) according to word frequen
(6)  if (terms q match the concepts in ontology database)
(7)      TermsWithOntologyConcept.add(q)
(8)  else
(9)      TermsWithoutOntologyConcept.add(q)
(10) if (TermsWithOntologyConcept.IsEmpty != true)
(11)     Decide the relation between terms.
(12) if (TermsWithoutOntologyConcept.IsEmpty != true)
(13)     foreach (term t in TermsWithoutOntologyConcept)
(14)         Nconceptmatching1.add(t)
(15) if (TermsWithOntologyConcept.IsEmpty == true &&
     TermsWithoutOntologyConcept.IsEmpty != true)
(16)     foreach (term t in TermsWithoutOntologyConcept)
(17)         Nconceptmatching2.add(t)
```

After applying semantic analysis on user request, semantic information is able to be used in the retrieval strategy. The pseudo-code of information retrieval algorithm is shown in Fig. 6.

Fig. 6

```
(1)   Input: query expression
(2)   Output: semantic concept set TermsWithOntologyConcept
(3)   if (TermsWithOntologyConcept.IsEmpty != true)
(4)      QueryExpansion(q1, q2, ..., qi)
(5)      // expanse concepts on synonym, semantic implying and extensi
(6)      Get retrieval results according to mapping in concepts and docs
(7)   returnNconceptmatching1
(8)      // returnNconceptmatching1 to users to adjust retrieval strategy
(9)   If (Nconceptmatching2.IsEmpty != true)
(10)     Calculate the similarity between (q1, q2, ..., qi) and (w1, w2, ..., v
(11)  Get retrieval results according to similarity and docs
(12)     return Nconceptmatching2
(13)     // returnNconceptmatching2 to users to adjust retrieval strategy
```

Pseudo-code for the information retrieval algorithm

Full size image

Experiment and results

The experimental design of the information retrieval model based on ontology

In order to evaluate the performance of the information retrieval model based on ontology, it is necessary to use ontology tools for modelling, such as Protégé [18] as an ontology modeling tool, ICTCLAS [19] as word segmentation tool, Jena [20] as semantic parsing tool, and Lucene as semantic indexing tool.

The data set contains 1000 scientific papers and papers from the IEEE digital library, which are used to extract the core concepts in the domain ontology. Then the final conceptualization system is established. The literature is divided into 10 groups. Each group contains 100 papers related to a query subject or key words (e.g., computer architecture and operating system). Therefore, 10 experts rank lists are available for retrieval.

The evaluation criterion considers the similarity of each paper towards every query word. For example, the mistaken sort term distance of the top neighboring papers is higher than the ones

of lowest papers. The formula below is for how to collect the distance within rank list R and R’:

$$P(t)=\frac{\sum_{i=1}^{n}[(n-i)\times dis(i)]}{\sum_{i=1}^{\lfloor n2 \rfloor}[(n-i)\times i]+\sum_{\lfloor n2 \rfloor}^{n}[(n-i)2]}$$

P(t)=∑i=1n[(n−i)×dis(i)]∑i=1⌊n2⌋[(n−i)×i]+∑⌊n2⌋n[(n−i)2]

(3)

Here, n represents the paper numbers in the rank list. The dis(i) represents the position distance for paper i in the rank list and expert rank list. $P(t)$ represents the distance between the two rank lists of the denominator specification.

Analysis of experimental results

The genetic algorithm with simulated annealing method is compared in relation to iteration numbers and average distance of the rank list. The result is shown in Fig. 7. The X-axis time is the number of iterations in two algorithms, and the Y-axis average distance is calculated by formula (3) which shows the difference of the ranking list with expert list. After iteration for 200 times, the average distance is close to overall optimal. The algorithm deduces the optimized weight combination of factors which are $w_{tit} = 3$, $w_{abs} = 2$, $w_{key} = 0.6$.

Fig. 7