

Module-7

Advanced Database Models

1. Temporal Database
2. Spatial Databases
3. Distributed Database
4. NoSQL Database
5. Mongo Database

Prepared by Dr. Parimala, SCORE

Temporal Database

- Temporal databases requires some aspect of time when organizing information
- Need for temporal database
 - Healthcare : Patients history of treatment
 - Insurance : claims and accident histories
 - Reservation systems: reservation date and time stored in applications like railway, hotel, airline and rental...,
 - Scientific databases : date and time of data collection

Representation of Time

- **Chronon**
 - Term used to describe minimal granularity of a particular application which depends upon the database designers
 - Eg., Some applications requires information per second but some may require information per minute.
- Reference point for measuring specific time events
 - Various calendars

SQL temporal data types

DATE, TIME, TIMESTAMP, INTERVAL, PERIOD

- **date**: four digits for the year (1--9999), two digits for the month (1--12), and two digits for the date (1--31).
- **time**: two digits for the hour, two digits for the minute, and two digits for the second, plus optional fractional digits.
- **timestamp**: the fields of **date** and **time**, with six fractional digits for the seconds field.
- **interval**: refers to a period of time (e.g., 2 days and 5 hours), without specifying a particular time when this period starts; could more accurately be termed a *span*.
- **Period**: an *anchored* time duration with a fixed starting point,
 - such as the 10-day period from January 1, 1999, to January 10, 1999, inclusive).

Point and Duration events

- **Point events**

- Typically associated with a single time point
- Time series data

For example, a bank deposit event may be associated with the timestamp when the deposit was made, or the total monthly sales of a product (fact) may be associated with a particular month (say, February 1999).

- **Duration events**

- Associated with specific time period
- Time period represented by start and end points

For example, an employee may have worked in a company from August 15, 1993, till November 20, 1998.

Types of temporal databases

- **Valid time relation**

- It is the time that the event occurred, or the period during which the fact was considered to be true in *the real world*.
- *The database that uses Valid time is called Valid time database*

- **Transaction time relation**

- associated time refers to the time when the information was actually stored in the database
- It is the value of the system time clock when the information is valid in *the system*
- A temporal database using this interpretation is called a transaction time database.

- **Bitemporal relation**

- Uses both valid time and transaction time

- **User-defined time**

- - User can define the semantics and program the applications appropriately

Temporal Database Concepts (cont'd.)

(a) EMP_VT

| | | | | | | |
|------|------------|--------|-----|----------------|------------|-----|
| Name | <u>Ssn</u> | Salary | Dno | Supervisor_ssn | <u>Vst</u> | Vet |
|------|------------|--------|-----|----------------|------------|-----|

DEPT_VT

| | | | | | |
|-------|------------|-----------|-------------|------------|-----|
| Dname | <u>Dno</u> | Total_sal | Manager_ssn | <u>Vst</u> | Vet |
|-------|------------|-----------|-------------|------------|-----|

Valid Time relation
Vst- Valid start time
Vet- Valid end time

(b) EMP_TT

| | | | | | | |
|------|------------|--------|-----|----------------|------------|-----|
| Name | <u>Ssn</u> | Salary | Dno | Supervisor_ssn | <u>Tst</u> | Tet |
|------|------------|--------|-----|----------------|------------|-----|

DEPT_TT

| | | | | | |
|-------|------------|-----------|-------------|------------|-----|
| Dname | <u>Dno</u> | Total_sal | Manager_ssn | <u>Tst</u> | Tet |
|-------|------------|-----------|-------------|------------|-----|

Transaction Time relation
Tst-Transaction start time
Tet- Transaction end time

Bitemporal
relation

(c) EMP_BT

| | | | | | | | | |
|------|------------|--------|-----|----------------|------------|-----|------------|-----|
| Name | <u>Ssn</u> | Salary | Dno | Supervisor_ssn | <u>Vst</u> | Vet | <u>Tst</u> | Tet |
|------|------------|--------|-----|----------------|------------|-----|------------|-----|

DEPT_BT

| | | | | | | | |
|-------|------------|-----------|-------------|------------|-----|------------|-----|
| Dname | <u>Dno</u> | Total_sal | Manager_ssn | <u>Vst</u> | Vet | <u>Tst</u> | Tet |
|-------|------------|-----------|-------------|------------|-----|------------|-----|

Incorporating Time in Relational Databases Using Tuple Versioning

- In EMP_VT, each tuple V represents a version of an employee's information that is valid (in the real world) only during the time period $[V.VST, V.VET]$, whereas in EMPLOYEE each tuple represents only the current state or current version of each employee.
- In EMP_VT, the current version of each employee typically has a special value, *now*, as its valid end time
- This special value, *now*, is a temporal variable that implicitly represents the **current time** as time progresses.
- The nontemporal EMPLOYEE relation would only include those tuples from the EMP_VT relation whose VET is *now*.

Tuple Version in Valid time relations

EMP_VT

| Name | <u>Ssn</u> | Salary | Dno | Supervisor_ssn | <u>Vst</u> | Vet |
|---------|------------|--------|-----|----------------|------------|------------|
| Smith | 123456789 | 25000 | 5 | 333445555 | 2002-06-15 | 2003-05-31 |
| Smith | 123456789 | 30000 | 5 | 333445555 | 2003-06-01 | Now |
| Wong | 333445555 | 25000 | 4 | 999887777 | 1999-08-20 | 2001-01-31 |
| Wong | 333445555 | 30000 | 5 | 999887777 | 2001-02-01 | 2002-03-31 |
| Wong | 333445555 | 40000 | 5 | 888665555 | 2002-04-01 | Now |
| Brown | 222447777 | 28000 | 4 | 999887777 | 2001-05-01 | 2002-08-10 |
| Narayan | 666884444 | 38000 | 5 | 333445555 | 2003-08-01 | Now |

...

DEPT_VT

| Dname | <u>Dno</u> | Manager_ssn | <u>Vst</u> | Vet |
|----------|------------|-------------|------------|------------|
| Research | 5 | 888665555 | 2001-09-20 | 2002-03-31 |
| Research | 5 | 333445555 | 2002-04-01 | Now |

...

Figure 2 - shows a few tuple versions in the valid-time relations EMP_VT and DEPT_VT.
There are two versions of Smith, three versions of Wong, one version of Brown, and one version of Narayan

Tuple Version in Valid time relations

- Whenever one or more attributes of an employee are updated, rather than actually overwriting the old values, as would happen in a nontemporal relation, the system will create a **new version** and close the current version by changing its VET to the end time

Eg., when the user issued the command to update the salary of Smith effective on June 1, 2003, to \$30000, the second version of Smith was created

- At the time of this update, the first version of Smith was the current version, with *now* as its VET, but after the update *now* was changed to May 31, 2003 (one less than June 1, 2003, in day granularity), to indicate that the version has become a closed or history version and that the new (second) version of Smith is now the current one.

Types of Updates

- (i) **Proactive**: It is applied to the database *before* it becomes effective in the real world
 - For example, the salary update of Smith may have been entered in the database on May 15, 2003, at 8:52:12 A.M., say, even though the salary change in the real world is effective on June 1, 2003.

- (ii) **Retroactive**: It is applied to the database *after* it became effective in the real world
 - Simultaneous: An update that is applied at the same time when it becomes effective

Deletion in Valid Time Relation

- Deleting an employee in a nontemporal database- *closing the current version* of the Employee
- For example, if Smith leaves the company effective January 19, 2004, then this would be applied by changing VET of the current version of Smith from *now* to 2004-01-19.
- There is no current version for Brown, because he presumably left the company on 2002-08-10 and was *logically deleted*.
- However, because the database is temporal, the old information on Brown is still there.

Insertion in valid time relation

- The operation to insert a new employee would correspond to *creating the first tuple version* for that employee, and making it the current version, with the VST being the effective (real world) time when the employee starts work
- Eg., the tuple on Narayan illustrates this, since the first version has not been updated yet.

Key for Valid time Temporal database

- Non temporal- Unique key

Eg., SSN in EMPLOYEE

- SSN is not unique in temporal database, So new relation key for EMP_VT is a combination of the nontemporal key and the valid start time attribute

PK (SSN, VST)

- If the nontemporal primary key value may change over time, it is important to have a unique surrogate key attribute, whose value never changes for each real world entity

Need for *Transaction Time relations*

- Valid time relations basically **keep track of the history** of changes as they become effective in the *real world*
- However, because updates, insertions, and deletions may be applied retroactively or proactively, there is no record of the actual *database state* at any point in time.
- If the actual database states are more important to an application, then one should use *transaction time relations*

Transaction Time relations

- In a transaction time database, whenever a change is applied to the database, the actual timestamp of the transaction that applied the change (insert, delete, or update) is recorded.

Eg., real-time stock trading or banking transactions.

- To convert the nontemporal database into a transaction time database, then the two relations EMPLOYEE and DEPARTMENT are converted into transaction time relations by adding the attributes TST (Transaction Start Time) and TET (Transaction End Time)
- In EMP_TI, each tuple v represents a *version* of an employee's information that was created at actual time v . TST and was (logically) removed at actual time v . TET (because the information was no longer correct)

Transaction Time relations -States

- In EMP_TI, the *currentversion* of each employee typically has a special value, *uc* (Until Changed), as its transaction end time, which indicates that the tuple represents correct information *until it is changed* by some other transaction
- A transaction time database has also been called a rollback database, because a user can logically roll back to the actual database state at any past point in time T by retrieving all tuple versions v whose transaction time period $[v.TST, V.TET]$ includes time point T .

Bitemporal Relations

- Tuples whose transaction end time TET is *uc* are the ones representing currently valid information, whereas tuples whose TET is an absolute timestamp are tuples that were valid until (just before) that timestamp.
- The transaction start time attribute TST in each tuple is the timestamp of the transaction that created that tuple.

Bitemporal Relation – Case study 1

- Consider a transaction (T) whose salary is updated to 5000 with effect from 1st October, 2024 which is stored in the database on 25th September, 2024. Update the above information in Bitemporal database
- Given: new salary=5000, VT=1st Oct,2024, TS(T)=25th Sep,2024
- Current state of database: v1

| Ssn | Name | Sal | Vst | Vet | Tst | Tet |
|-----|------|------|-----------|-----|-----------|-----|
| 101 | xyz | 2000 | 1/10/2021 | now | 6/10/2021 | uc |

- After Updation two records are added (v2 and v3)

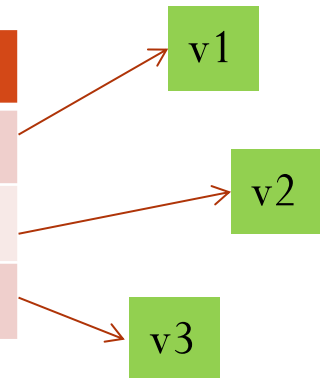
Bitemporal relation

| Step 1: v2 | Step 2: v3 |
|------------------|--------------------|
| v2.ssn= v1.ssn | V3.ssn=v1.ssn |
| v2. name=v1.name | V3.name=v1.name |
| v2.sal = v1.sal | V3.sal= new salary |
| v2. vst= v1.vst | V3.vst= VT |
| v2.vet= VT-1 | V3.vet=now |
| v2. tst= TS(T) | V3.tst=TS(T) |
| v2.tet= uc | V3.tet=uc |

Step 1: Update v2 record
 Step 2: Update v3 record
 Step 3: Update v1.tet=TS(T)

After Updation

| Ssn | Name | Sal | Vst | Vet | Tst | Tet |
|-----|------|------|-----------|-----------|-----------|---------|
| 101 | xyz | 2000 | 1/10/2021 | now | 6/10/2021 | 25/9/24 |
| 101 | xyx | 2000 | 1/10/2021 | 31/9/2024 | 25/9/24 | Uc |
| 101 | xyz | 5000 | 1/10/2024 | now | 25/9/24 | Uc |



Bitemporal relation- Snapshot EMP database

EMP_BT

| Name | <u>Ssn</u> | Salary | Dno | Supervisor_ssn | <u>Vst</u> | Vet | <u>Tst</u> | Tet |
|---------|------------|--------|-----|----------------|------------|------------|----------------------|---------------------|
| Smith | 123456789 | 25000 | 5 | 333445555 | 2002-06-15 | Now | 2002-06-08, 13:05:58 | 2003-06-04,08:56:12 |
| Smith | 123456789 | 25000 | 5 | 333445555 | 2002-06-15 | 2003-05-31 | 2003-06-04, 08:56:12 | uc |
| Smith | 123456789 | 30000 | 5 | 333445555 | 2003-06-01 | Now | 2003-06-04, 08:56:12 | uc |
| Wong | 333445555 | 25000 | 4 | 999887777 | 1999-08-20 | Now | 1999-08-20, 11:18:23 | 2001-01-07,14:33:02 |
| Wong | 333445555 | 25000 | 4 | 999887777 | 1999-08-20 | 2001-01-31 | 2001-01-07, 14:33:02 | uc |
| Wong | 333445555 | 30000 | 5 | 999887777 | 2001-02-01 | Now | 2001-01-07, 14:33:02 | 2002-03-28,09:23:57 |
| Wong | 333445555 | 30000 | 5 | 999887777 | 2001-02-01 | 2002-03-31 | 2002-03-28, 09:23:57 | uc |
| Wong | 333445555 | 40000 | 5 | 888667777 | 2002-04-01 | Now | 2002-03-28, 09:23:57 | uc |
| Brown | 222447777 | 28000 | 4 | 999887777 | 2001-05-01 | Now | 2001-04-27, 16:22:05 | 2002-08-12,10:11:07 |
| Brown | 222447777 | 28000 | 4 | 999887777 | 2001-05-01 | 2002-08-10 | 2002-08-12, 10:11:07 | uc |
| Narayan | 666884444 | 38000 | 5 | 333445555 | 2003-08-01 | Now | 2003-07-28, 09:25:37 | uc |

Spatial Database

- **Spatial databases** provide concepts for databases that keep track of objects in a multidimensional space. spatial database stores objects that have spatial characteristics that describe them.
- For eg., cartographic databases that store maps include two dimensional spatial descriptions of their objects-from countries and states to rivers, cities, roads, seas, and so on.
- These applications are also known as Geographical Information Systems (GIS), and are used in areas such as environmental, emergency, and battle management.
- Other databases, such as meteorological databases for weather information, are three-dimensional, since temperatures and other meteorological information are related to three-dimensional spatial points
- The spatial relationships among the objects are important, and they are often needed when querying the database.

Spatial Data Types

- The basic extensions needed are to include two dimensional geometric concepts, such as points, lines and line segments, circles, polygons, and arcs, in order to specify the spatial characteristics of objects
- Spatial data types
 - Map data
 - Geographic or spatial features of objects in a map
 - Attribute data
 - Descriptive data associated with map features
 - Image data
 - Satellite images

Spatial Operations

- Spatial operations are needed to operate on the objects' spatial characteristics
- For example,
 - To compute Spatial **distance between** two objects
 - Spatial **Boolean conditions** to check whether two objects spatially overlap.

Types of spatial objects

- A description of the spatial positions of many types of objects would be needed.
- Some objects may have **static spatial characteristics**, such as streets and highways, water pumps, (for fire control), police stations, fire stations, and hospitals.
- Other objects have **dynamic spatial characteristics** that change over time, such as police vehicles, ambulances, or fire trucks.

Spatial Queries

Range query: Finds the objects of a particular type that are within a given spatial area or within a particular distance from a given location.

For eg., finds all hospitals within the Dallas city area, or finds all ambulances within five miles of an accident location

Nearest neighbor query: Finds an object of a particular type that is closest to a given location.

For eg., finds the police car that is closest to a particular location

Spatial joins or overlays: Typically joins the objects of two types based on some spatial condition, such as the objects intersecting or overlapping spatially or being within a certain distance of one another.

For example, finds all cities that fall on a major highway or finds all homes that are within two miles of a lake.

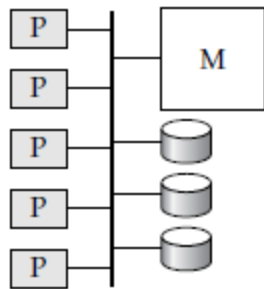
Spatial Indexing

- To execute the spatial query efficiently, spatial indexing can be used.
- Spatial data indexing
 - Grid files
 - R-trees
 - Spatial join index
- Spatial data mining techniques
 - Spatial classification
 - Spatial association
 - Spatial clustering

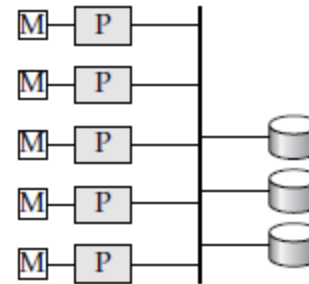
Distributed Database Concepts

- A transaction can be executed by multiple networked computers in a unified manner.
- A **distributed database (DDB)** processes Unit of execution (a transaction) in a distributed manner.
- A distributed database (DDB) can be defined as
 - A distributed database (DDB) is a collection of multiple logically related database distributed over a computer network, and a distributed database management system as a software system that manages a distributed database while making the distribution transparent to the user.

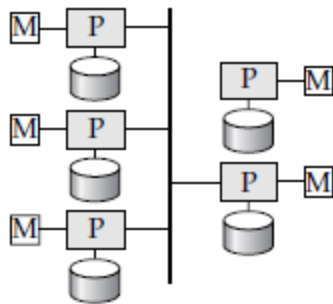
Parallel database architectures



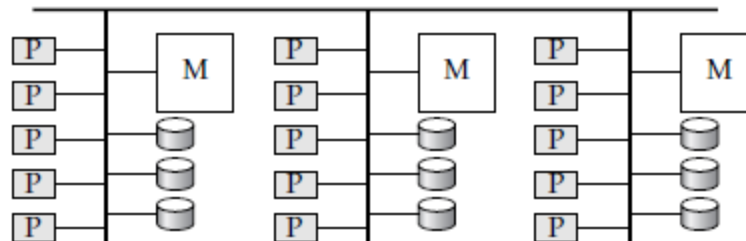
(a) shared memory



(b) shared disk

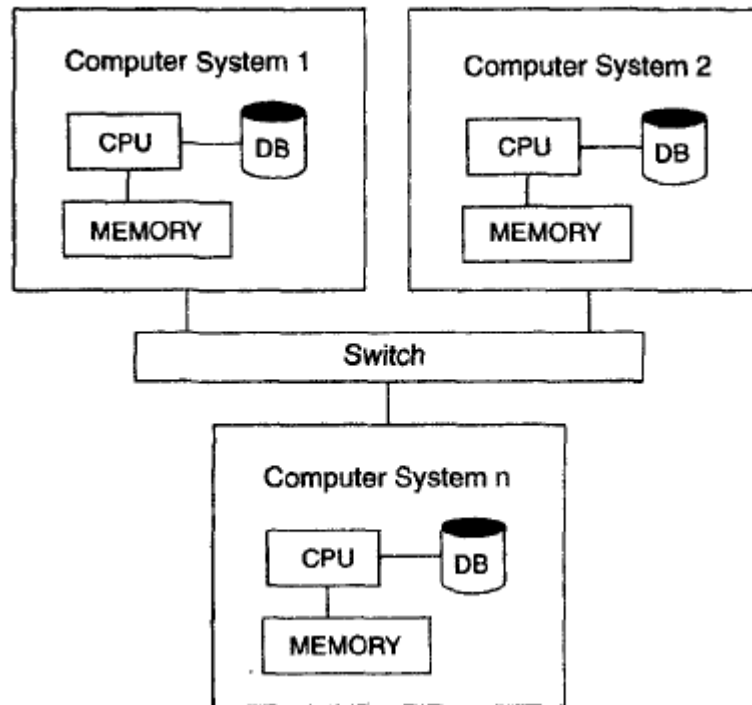


(c) shared nothing



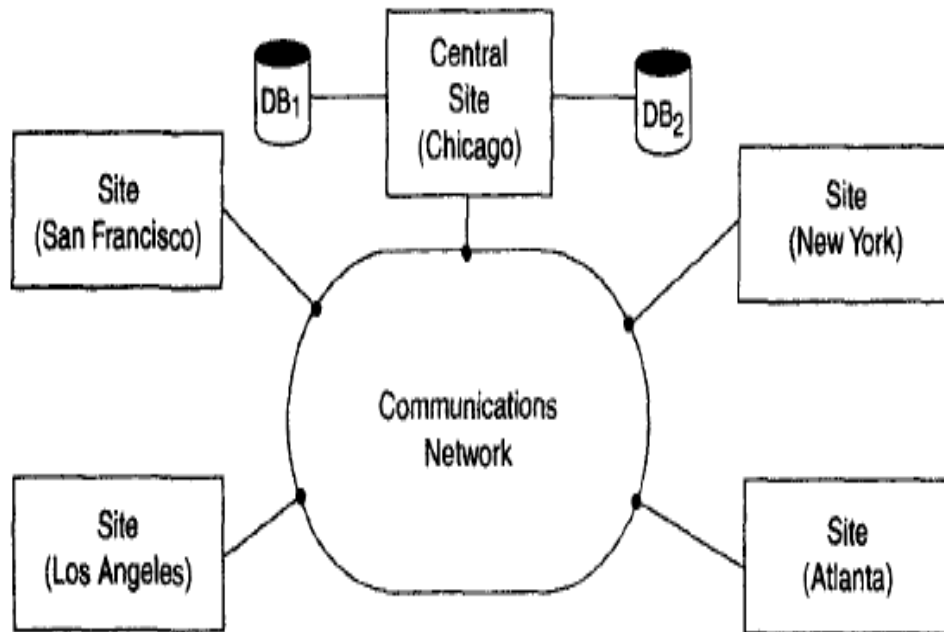
(d) hierarchical

Different Database Architecture



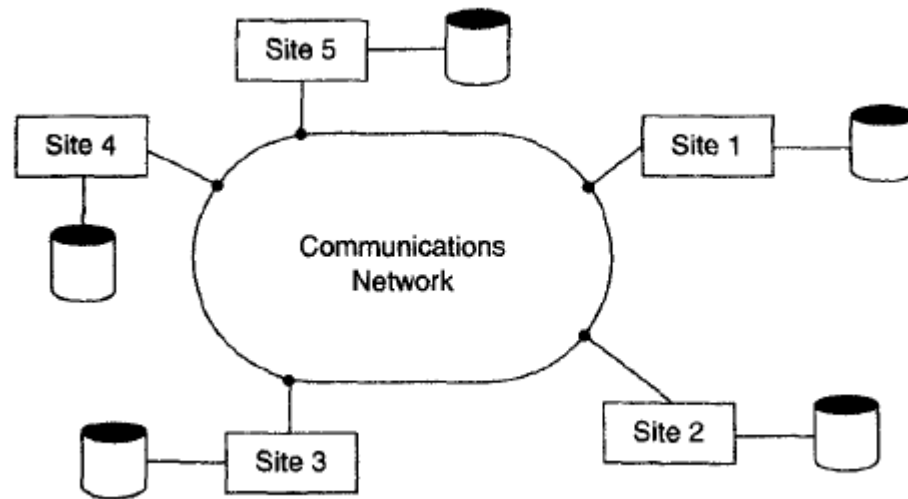
Shared nothing
Architecture - every processor has its own primary and secondary (disk) memory, no common memory exists, and the processors communicate over a high-speed interconnection network (bus or switch).

Different Database Architecture



A networked architecture with a centralized database at one of the sites.

Different Database Architecture



A truly distributed database architecture.

Difference between shared-nothing parallel databases and distributed databases

Distributed databases are typically **geographically separated**,

Distributed database system have **local and global transactions**.

A **local transaction** is one that accesses data only from sites where the transaction was initiated.

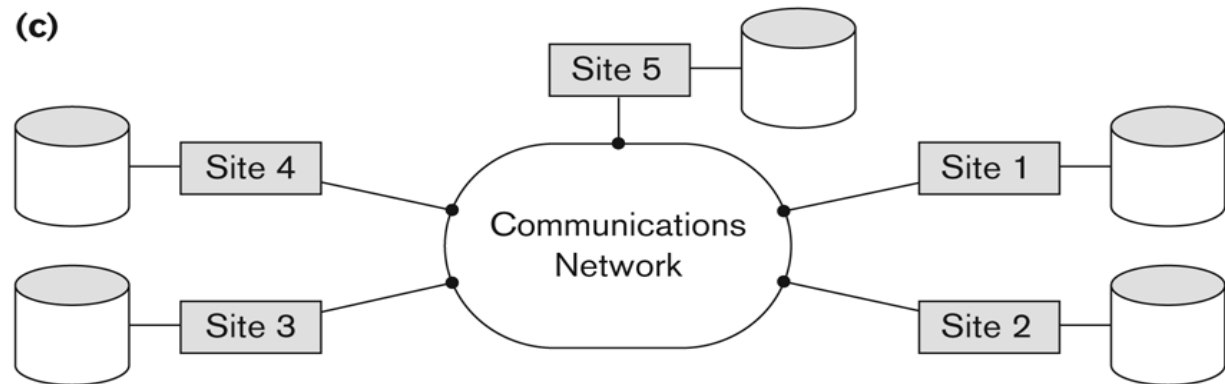
A **global transaction**, on the other hand, is one that either accesses data in a site different from the one at which the transaction was initiated, or accesses data in several different sites.

Advantage-1: Management of Distributed Data With Different Levels of transparency

This refers to the physical placement of data (files, relations, etc.) which is not known to the user (distribution transparency).

Figure 25.1

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.

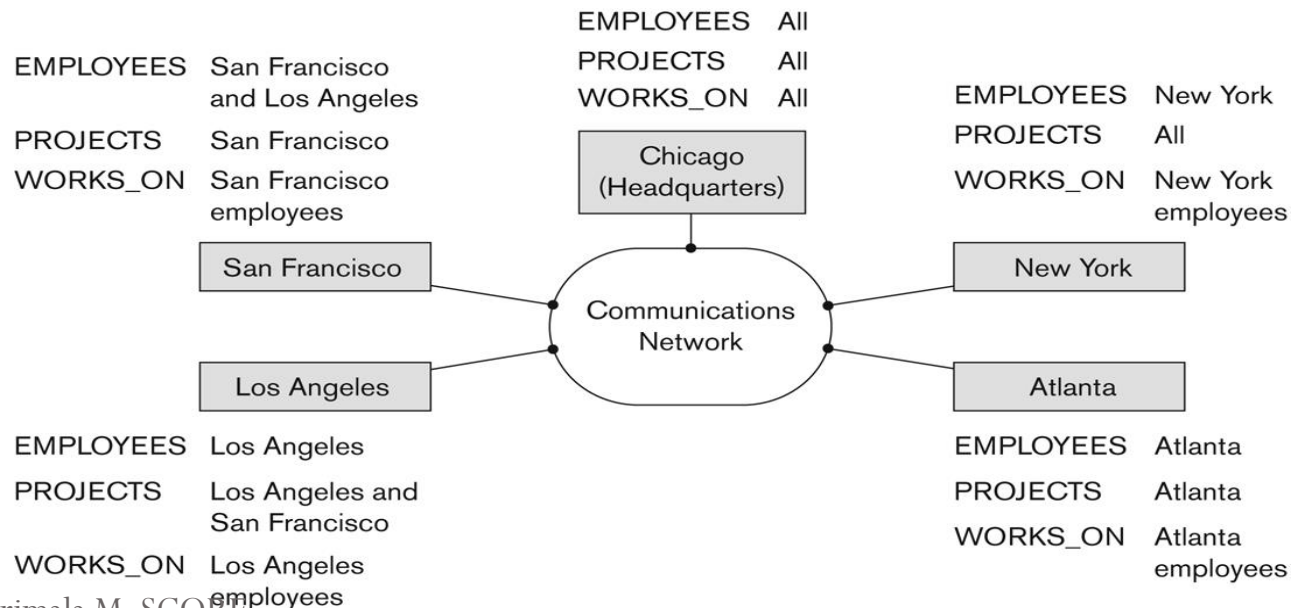


Advantage-1: Management of Distributed Data With Different Levels of Transparency

- The EMPLOYEE, PROJECT, and WORKS_ON tables may be fragmented horizontally and stored with possible replication as shown below.

Figure 25.2

Data distribution and replication among distributed databases.



Levels of Transparency

There are three levels of Transparency.

- (i) Distribution and Network transparency
- (ii) Replication transparency
- (iii) Fragmentation transparency

(i) Distribution and Network transparency:

- Users do not have to worry about operational details of the network.
 - There is Location transparency, which refers to freedom of issuing command from any location without affecting its working.
 - Then there is Naming transparency, which allows access to any names object (files, relations, etc.) from any location.

Levels of Transparency

(ii) Replication transparency:

- It allows to store copies of a data at multiple sites as shown in the above diagram.
- This is done to minimize access time to the required data.

(iii) Fragmentation transparency:

- Allows to fragment a relation horizontally (create a subset of tuples of a relation) or vertically (create a subset of columns of a relation).

Advantage -2: Increased Reliability and Availability

- Reliability refers to system live time, that is, system is running efficiently most of the time. Availability is the probability that the system is continuously available (usable or accessible) during a time interval.
- A distributed database system has multiple nodes (computers) and if one fails then others are available to do the job.

Advantage -3 : Improved Performance

Advantage – 4: Easier Expansion (Scalability)

Improved performance:

- A distributed DBMS fragments the database to keep data closer to where it is needed most.
- This reduces data management (access and modification) time significantly.

● **Easier expansion (scalability):**

- Allows new nodes (computers) to be added anytime without chaining the entire configuration.

Data Fragmentation

- **Data Fragmentation**
 - Split a relation into logically related and correct parts. A relation can be fragmented in two ways:
 - **Horizontal Fragmentation**
 - **Vertical Fragmentation**

Horizontal Data Fragmentation

- **Horizontal fragmentation**

- It is a horizontal subset of a relation which contain those of **tuples which satisfy selection conditions**.
- Consider the Employee relation with selection condition ($DNO = 5$). All tuples satisfy this condition will create a subset which will be a horizontal fragment of Employee relation.
- A selection condition may be composed of several conditions connected by AND or OR.
- Derived horizontal fragmentation: It is the partitioning of a primary relation to other secondary relations which are related with Foreign keys.

Horizontal Data Fragmentation

- **Horizontal fragmentation**

- Each horizontal fragment on a relation can be specified by a $\sigma_{C_i}(R)$ operation in the relational algebra.
- Complete horizontal fragmentation
- A set of horizontal fragments whose conditions C_1, C_2, \dots, C_n include all the tuples in R - that is, every tuple in R satisfies $(C_1 \text{ OR } C_2 \text{ OR } \dots \text{ OR } C_n)$.
- Disjoint complete horizontal fragmentation: No tuple in R satisfies $(C_i \text{ AND } C_j)$ where $i \neq j$.
- To reconstruct R from horizontal fragments a UNION is applied.

Horizontal Data Fragmentation

(a)

| EMPID5 | FNAME | MINIT | LNAME | <u>SSN</u> | SALARY | SUPERSSN | DNO |
|--------|----------|-------|---------|------------|--------|-----------|-----|
| | John | B | Smith | 123456789 | 30000 | 333445555 | 5 |
| | Franklin | T | Wong | 333445555 | 40000 | 888665555 | 5 |
| | Ramesh | K | Narayan | 666884444 | 38000 | 333445555 | 5 |
| | Joyce | A | English | 453453453 | 25000 | 333445555 | 5 |

| DEP5 | DNAME | <u>DNUMBER</u> | MGRSSN | MGRSTARTDATE |
|------|----------|----------------|-----------|--------------|
| | Research | 5 | 333445555 | 1988-05-22 |

| DEP5_LOCS | <u>DNUMBER</u> | <u>LOCATION</u> |
|-----------|----------------|-----------------|
| | 5 | Bellaire |
| | 5 | Sugarland |
| | 5 | Houston |

| WORKS_ON5 | <u>ESSN</u> | <u>PNO</u> | HOURS |
|-----------|-------------|------------|-------|
| | 123456789 | 1 | 32.5 |
| | 123456789 | 2 | 7.5 |
| | 666884444 | 3 | 40.0 |
| | 453453453 | 1 | 20.0 |
| | 453453453 | 2 | 20.0 |
| | 333445555 | 2 | 10.0 |
| | 333445555 | 3 | 10.0 |
| | 333445555 | 10 | 10.0 |
| | 333445555 | 20 | 10.0 |

| PROJ5 | PNAME | <u>PNUMBER</u> | PLOCATION | DNUM |
|-------|-----------|----------------|-----------|------|
| | Product X | 1 | Bellaire | 5 |
| | Product Y | 2 | Sugarland | 5 |
| | Product Z | 3 | Houston | 5 |

Data at Site 2

Relation fragments at site 2
corresponding to department 5

Vertical Data Fragmentation

- **Vertical fragmentation**

- It is a subset of a relation which is created by a subset of columns. Thus a vertical fragment of a relation will contain values of selected columns. There is no selection condition used in vertical fragmentation.
- Consider the Employee relation. A vertical fragment of can be created by keeping the values of Name, Bdate, Sex, and Address.
- Because there is no condition for creating a vertical fragment, each fragment must include the primary key attribute of the parent relation Employee. In this way all vertical fragments of a relation are connected.

Vertical Data Fragmentation

Vertical fragmentation

- A vertical fragment on a relation can be specified by a $\Pi_{L_i}(R)$ operation in the relational algebra.
- Complete vertical fragmentation
- A set of vertical fragments whose projection lists L_1, L_2, \dots, L_n include all the attributes in R but share only the primary key of R . In this case the projection lists satisfy the following two conditions:
 - $L_1 \cup L_2 \cup \dots \cup L_n = \text{ATTRS}(R)$
 - $L_i \cap L_j = \text{PK}(R)$ for any i, j , where $\text{ATTRS}(R)$ is the set of attributes of R and $\text{PK}(R)$ is the primary key of R .
- To reconstruct R from complete vertical fragments a OUTER UNION is applied.

Mixed Data Fragmentation

- **Representation**

- **Mixed (Hybrid) fragmentation**

- A combination of Vertical fragmentation and Horizontal fragmentation.
 - This is achieved by SELECT-PROJECT operations which is represented by $\Pi_{L_i}(\sigma_{C_i}(R))$.
 - If $C = \text{True}$ (Select all tuples) and $L \neq \text{ATTRS}(R)$, we get a vertical fragment, and if $C \neq \text{True}$ and $L \neq \text{ATTRS}(R)$, we get a mixed fragment.
 - If $C = \text{True}$ and $L = \text{ATTRS}(R)$, then R can be considered a fragment.

Data Fragmentation, Replication and Allocation

- **Fragmentation schema**

- A definition of a set of fragments (horizontal or vertical or horizontal and vertical) that includes all attributes and tuples in the database that satisfies the condition that the whole database can be reconstructed from the fragments by applying some sequence of JOIN (or OUTER JOIN) and UNION operations.

- **Allocation schema**

- It describes the distribution of fragments to sites of distributed databases. It can be fully or partially replicated or can be partitioned.

Data Fragmentation, Replication and Allocation

- **Data Replication**

- Database is replicated to all sites.
- In full replication the entire database is replicated and in partial replication some selected part is replicated to some of the sites.
- Data replication is achieved through a replication schema.

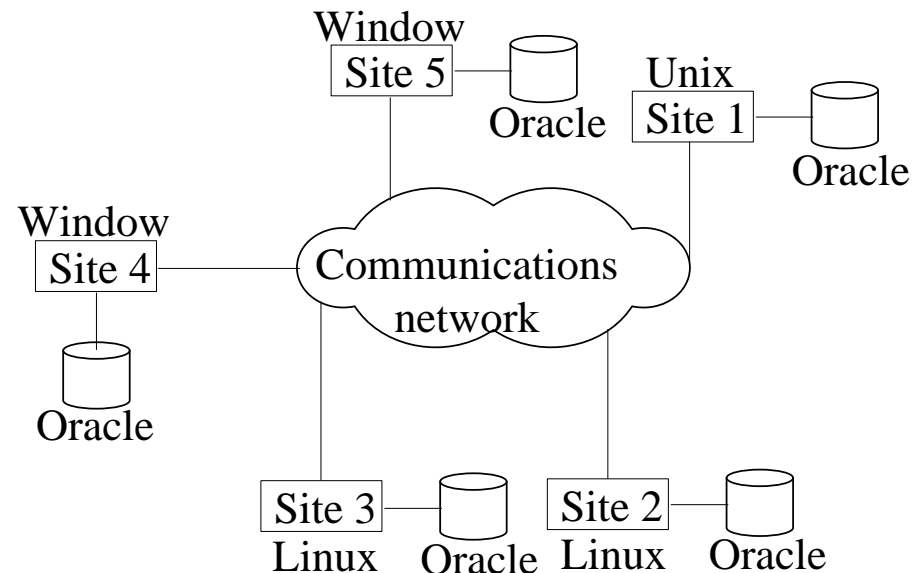
- **Data Distribution (Data Allocation)**

- This is relevant only in the case of partial replication or partition.
- The selected portion of the database is distributed to the database sites.

Types of Distributed Database Systems

- **Homogeneous**

- All sites of the database system have identical setup, i.e., same database system software.
- The underlying operating system may be different.
 - For example, all sites run Oracle or DB2, or Sybase or some other database system.
- The underlying operating systems can be a mixture of Linux, Window, Unix, etc.

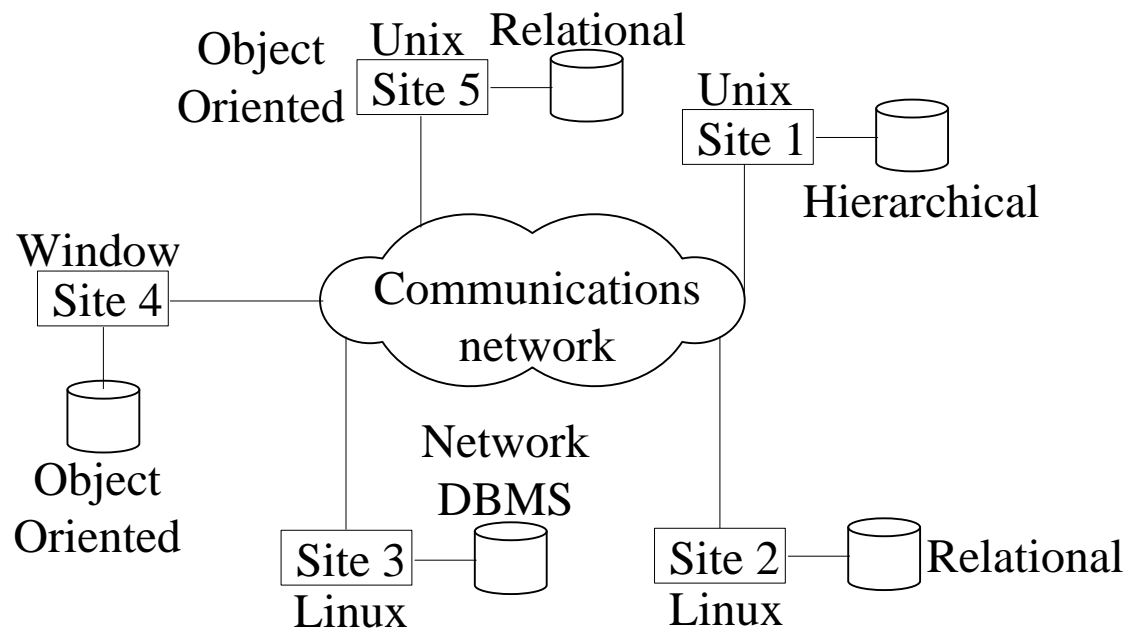


Types of Distributed Database Systems

- **Heterogeneous**
- **Federated:** Each site may run different database system but the data access is managed through a single conceptual schema.
 - This implies that the degree of local autonomy is minimum. Each site must adhere to a centralized access policy. There may be a global schema.

Types of Distributed Database

- **Multidatabase:** There is no one conceptual global schema. For data access a schema is constructed dynamically as needed by the application software.



Types of Distributed Database Systems

- Federated Database Management Systems Issues
 - Differences in data models:
 - Relational, Objected oriented, hierarchical, network, etc.
 - Differences in constraints:
 - Each site may have their own data accessing and processing constraints.
 - Differences in query language:
 - Some site may use SQL, some may use SQL-89, some may use SQL-92, and so on.

NoSQL

- NOSQL
 - Not only SQL
 - SQL systems offer many features (not everyone will use) and restrictive
- Most NOSQL systems are distributed databases or distributed storage systems
 - Focus on semi-structured data storage, high performance, availability, data replication, and scalability

Introduction (cont'd.)

- NOSQL systems focus on storage of “big data”
- Typical applications that use NOSQL
 - Social media
 - Web links
 - User profiles
 - Marketing and sales
 - Posts and tweets
 - Road maps and spatial data
 - Email

NoSQL databases

- BigTable
 - Used in Gmail, Google Maps, and Web site indexing
 - Google's proprietary NOSQL system
 - Column-based or wide column store
- DynamoDB (Amazon)
 - Key-value data store
 - key-tuple or key-object data stores
- Cassandra (Facebook)
 - Uses concepts from both key-value store and column-based systems

Characteristics of NoSQL

Scalability:

- Horizontal scalability is generally used, where the distributed system is expanded by adding more nodes for data storage and processing as the volume of data grows.
- Vertical scalability, on the other hand, refers to expanding the storage and computing power of existing nodes.

Availability, replication, and eventual consistency:

- Data is replicated over two or more nodes in a transparent manner, so that if one node fails, the data is still available on other nodes.
- Replication improves data availability and can also improve read performance, because read requests can often be serviced from any of the replicated data nodes.
- Many NOSQL applications do not require serializable consistency, so more relaxed forms of consistency known as eventual consistency are used

Characteristics of NoSQL

Replication models

Master-slave: all write operations must be applied to the master copy and then propagated to the slave copies, usually using eventual consistency (the slave copies will eventually be the same as the master copy)

Master-Master: allows reads and writes at any of the replicas but may not guarantee that reads at nodes that store different copies see the same value

Characteristics of NoSQL

Sharding of files

- Also known as horizontal partitioning
- This serves to distribute the load of accessing the file records to multiple nodes.
- The combination of sharding the file records and replicating the shards works in tandem to improve load balancing as well as data availability.

High performance data access

- Finding individual records or objects (data items) from among the millions of data records or objects in a file.
- Two techniques used for data access is hashing or range partitioning on object keys

Characteristics of NoSQL

- NOSQL characteristics related to data models and query languages
 - Schema not required - The flexibility of not requiring a schema is achieved in many NOSQL systems by allowing semi-structured, self-describing data
 - Less powerful query languages
 - CRUD- Create, Read, Update, Delete
 - SCRUD- CRUD with Search operations
 - Versioning - provide storage of multiple versions of the data items, with the timestamps of when the data version was created

Types of NoSQL

+ **Document-based NOSQL systems**

These systems store data in the form of documents using well-known formats, such as JSON (JavaScript Object Notation). Documents are accessible via their document id, but can also be accessed rapidly using other indexes. Eg., MongoDB

+ **NoSQL key-value stores**

These systems have a simple data model based on fast access by the key to the value associated with the key; the value can be a record or an object or a document or even have a more complex data structure.

Eg., Oracle NoSQL, CouchDB

+ **Column-based or wide column NOSQL systems**

These systems partition a table by column into column families (a form of vertical partitioning) where each column family is stored in its own files. They also allow versioning of data values.

Eg., Cassandra, Hbase, Amazon DynamoDB

Other NoSQL databases

+ **Graph-based NOSQL systems:** Data is represented as graphs, and related nodes can be found by traversing the edges using path expressions. Eg., Neo4j

+ **Hybrid NOSQL systems:** These systems have characteristics from two or more of the above four categories.

- Object databases
- XML databases

The CAP Theorem

- Various levels of consistency among replicated data items
 - Enforcing serializability the strongest form of consistency
 - High overhead – can reduce read/write operation performance
- CAP theorem
 - According to Eric Brewer, three properties of Distributed system are
 - Consistency
 - Availability
 - Partition tolerance
 - Not possible to guarantee all three simultaneously
 - In distributed system with data replication

The CAP Theorem (cont'd.)

- Designer can choose two of three to guarantee
 - Weaker consistency level is often acceptable in NOSQL distributed data store
 - Guaranteeing availability and partition tolerance more important
 - Eventual consistency often adopted

Advantages of NoSQL

- Cheap and easy to implement (open source)
- Data are replicated to multiple nodes (more availability and fault tolerance)
- Easy to distribute
- Don't require a schema

NoSQL does not provide..

- Joins
- Group by
- ACID transactions
- SQL
- Integration with applications that are based on SQL

When NoSQL is required

- NoSQL Data storage systems makes sense for applications that need to deal with very very large semi-structured data
 - ❑ Log Analysis
 - ❑ Social Networking Feeds
- Most of us work on organizational databases, which are not that large and have low update/query rates
 - ❑ regular relational databases are THE correct solution for such applications

Document-Based NOSQL Systems and MongoDB

- Document stores
 - Collections of similar documents
- Individual documents resemble complex objects or XML documents
 - Documents are self-describing
 - Can have different data elements
- Documents can be specified in various formats
 - XML
 - JSON

MongoDB Data Model

- Documents stored in binary JSON (BSON) format
- Individual documents stored in a collection
- Example command
 - First parameter specifies name of the collection
 - Collection options include limits on size and number of documents

```
db.createCollection("project", { capped : true, size : 1310720, max : 500 } )  
_id
```

MongoDB Data Model (cont'd.)

- A collection does not have a schema
 - Structure of the data fields in documents chosen based on how documents will be accessed
 - User can choose normalized or denormalized design
- Document creation using insert operation

- Document deletion using remove operation
`db.<collection_name>.insert(<document(s)>)`

`db.<collection_name>.remove(<condition>)`

(a) project document with an array of embedded workers:

```
{
  _id:          "P1",
  Pname:        "ProductX",
  Plocation:    "Bellaire",
  Workers: [
    { Ename: "John Smith",
      Hours: 32.5
    },
    { Ename: "Joyce English",
      Hours: 20.0
    }
  ]
};
```

(b) project document with an embedded array of worker ids:

```
{
  _id:          "P1",
  Pname:        "ProductX",
  Plocation:    "Bellaire",
  WorkerIds:    [ "W1", "W2" ]
}

{ _id:          "W1",
  Ename:        "John Smith",
  Hours:        32.5
}

{ _id:          "W2",
  Ename:        "Joyce English",
  Hours:        20.0
}
```

Figure 24.1 (continues)

Example of simple documents in
MongoDB (a) Denormalized document
design with embedded subdocuments (b)
Embedded array of document references

Figure 24.1 (cont'd.) Example of simple documents in MongoDB (c) Normalized documents (d) Inserting the documents in Figure 24.1(c) into their collections

(c) normalized project and worker documents (not a fully normalized design for M:N relationships):

```
{
  _id:          "P1",
  Pname:        "ProductX",
  Plocation:    "Bellaire"
}
{
  _id:          "W1",
  Ename:        "John Smith",
  ProjectId:    "P1",
  Hours:        32.5
}

{
  _id:          "W2",
  Ename:        "Joyce English",
  ProjectId:    "P1",
  Hours:        20.0
}
```

(d) inserting the documents in (c) into their collections "project" and "worker":

```
db.project.insert( { _id: "P1", Pname: "ProductX", Plocation: "Bellaire" } )
db.worker.insert( [ { _id: "W1", Ename: "John Smith", ProjectId: "P1", Hours: 32.5 },
                    { _id: "W2", Ename: "Joyce English", ProjectId: "P1",
                      Hours: 20.0 } ] )
```

Mongo DB

- MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

Database

- Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

Mongo DB

Collection

- Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema.

Document

- A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

MongoDB Vs RDBMS

| RDBMS | MongoDB |
|-------------|--|
| Database | Database |
| Table | Collection |
| Tuple/Row | Document |
| column | Field |
| Table Join | Embedded Documents |
| Primary Key | Primary Key (Default key _id provided by mongodb itself) |

Create Database

use DATABASE_NAME is used to create database. The command will create a new database if it doesn't exist, otherwise it will return the existing database.

use DATABASE_NAME

Eg., >use sample- creates sample database

>db – displays the currently created database

>show dbs – displays all the created database

Note: If the created database 'sample' is not present in list, then we need to insert at least one document into it.

In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

Drop database

- **db.dropDatabase()** command is used to drop a existing database.
- ```
>use sample
```

```
switched to db mydb
```

```
>db.dropDatabase()
```

```
>{ "dropped" : "sample", "ok" : 1 } >
```

# Create collection

| Parameter | Type     | Description                                               |
|-----------|----------|-----------------------------------------------------------|
| Name      | String   | Name of the collection to be created                      |
| Options   | Document | (Optional) Specify options about memory size and indexing |

- **db.createCollection(name, options)** is used to create collection.

>use sample

switched to db sample

>db.createCollection("project")

{ "ok" : 1 }

# Create collection

```
db.createCollection("project", { capped : true, size : 1310720, max : 500 })
```

| Field       | Type    | Description                                                                                                                                                                                                                                           |
|-------------|---------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| capped      | Boolean | (Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. <b>If you specify true, you need to specify size parameter also.</b> |
| autoIndexId | Boolean | (Optional) If true, automatically create index on _id field.s Default value is false.                                                                                                                                                                 |
| size        | number  | (Optional) Specifies a maximum size in bytes for a capped collection. <b>If capped is true, then you need to specify this field also.</b>                                                                                                             |
| max         | number  | (Optional) Specifies the maximum number of documents allowed in the capped collection.                                                                                                                                                                |

# Delete collection

**db.collection\_name.drop()** is used to drop a collection from the database.

```
>use sample
```

```
switched to db sample
```

```
>db.project.drop()
```

```
true
```

drop() method will return true, if the selected collection is dropped successfully, otherwise it will return false.

# Insert document

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

```
>db.project.insert(
```

```
[
```

```
 {Pno: 101, Pname: "XYZ", Ploc: ["Chennai","Bangalore",
 Vellore"]},
```

```
 { Pno: 102, Pname: "UVW", Ploc: ["Chennai","Bangalore"],
 Dept: [{Id: 1, dname: "Accounts"}]
```

```
 }
```

```
])
```

# Find()

**find()** method will display all the documents in a non-structured way.

```
>db.project.find()
```

Pretty() -To display the results in a formatted way

```
>db.project.find().pretty()
```

**findOne()** method - returns only one document.

//similar to where clause in SQL

```
> db.project.findOne({pname: "XYZ"})
```



# Relational operators

| Operation           | Syntax                   | Example                                          | RDBMS Equivalent             |
|---------------------|--------------------------|--------------------------------------------------|------------------------------|
| Equality            | {<key>: {\$eq:<value>}}  | db.mycol.find({"by":"tutorials point"}).pretty() | where by = 'tutorials point' |
| Less Than           | {<key>: {\$lt:<value>}}  | db.mycol.find({"likes": {\$lt:50}}).pretty()     | where likes < 50             |
| Less Than Equals    | {<key>: {\$lte:<value>}} | db.mycol.find({"likes": {\$lte:50}}).pretty()    | where likes <= 50            |
| Greater Than        | {<key>: {\$gt:<value>}}  | db.mycol.find({"likes": {\$gt:50}}).pretty()     | where likes > 50             |
| Greater Than Equals | {<key>: {\$gte:<value>}} | db.mycol.find({"likes": {\$gte:50}}).pretty()    | where likes >= 50            |
| Not Equals          | {<key>: {\$ne:<value>}}  | db.mycol.find({"likes": {\$ne:50}}).pretty()     | where likes != 50            |

# Relational operators

|                        |                                                                                       |                                                                              |                                                                                        |
|------------------------|---------------------------------------------------------------------------------------|------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| Values in an array     | <code>{&lt;key&gt;:{\$in:[&lt;value1&gt;,&lt;value2&gt;,.....&lt;valueN&gt;]}}</code> | <code>db.mycol.find({"name":{\$in:["Raj", "Ram", "Raghu"]}}).pretty()</code> | Where name matches any of the value in :<br>["Raj", "Ram", "Raghu"]                    |
| Values not in an array | <code>{&lt;key&gt;:{\$nin:&lt;value&gt;}}</code>                                      | <code>db.mycol.find({"name":{\$nin:["Ramu", "Raghav"]}}).pretty()</code>     | Where name values is not in the array :<br>["Ramu", "Raghav"] or, doesn't exist at all |

# AND operator

```
>db.project.find({ $and: [{<key1>:<value1>}, {
<key2>:<value2>}] })
```

```
➤ db.project.find({ $and:[{Pno:1}, {"title": "XYZ"}] }).
pretty()
```

# OR operator

```
>db.project.find(
 {
 $or: [
 {key1: value1}, {key2:value2}
]
 }
)
```

```
db. Project.find({ $or:[{pno:2} , {“pname”:”UVW”}] }).pretty()
```

```
>db.mycol.find({ "likes": { $gt:10}, $or: [{ "by": “ABC”}, { "title":
"MongoDB Overview" }] }).pretty()
```

# NOT operator

```
>db.COLLECTION_NAME.find({ $NOT: [{key1: value1},
{key2:value2}] }).pretty()
```

Following example will retrieve the document(s) whose age is not greater than 25

```
> db.empDetails.find({ "Age": { $not: { $gt: "25" } } })
```

# Update

```
>db.mycol.update({'title':'MongoDB
Overview'}, { $set: {'title':'updated MongoDB Tutorial'}})
```

**findOneAndUpdate()** method updates the values in the existing document based on condition

Eg., updates the age and email values of the document with name 'Radhika'.

```
> db.empDetails.findOneAndUpdate({First_Name:
'Radhika'}, { $set: { Age: '30',e_mail:
'radhika_newemail@gmail.com'}})
```

# Update

- updateMany() method updates all the documents that matches the given filter.
- ```
> db.empDetails.updateMany( {Age:{ $gt: "25" }}, { $set: { Age: '00'}} )
```

Delete

```
>db.mycol.remove({'title':'MongoDB Overview'})  
>db.COLLECTION_NAME.remove(DELETION_CRITERIA)  
//MongoDB will delete whole documents from the collection.  
This is equivalent of SQL's truncate command.
```

If we want to delete only the first record, then set **justOne** parameter in **remove()** method.

```
db.COLLECTION_NAME.remove(DELETION_CRITERIA,1)  
>db.mycol.remove({'title':'MongoDB Overview'},1)
```

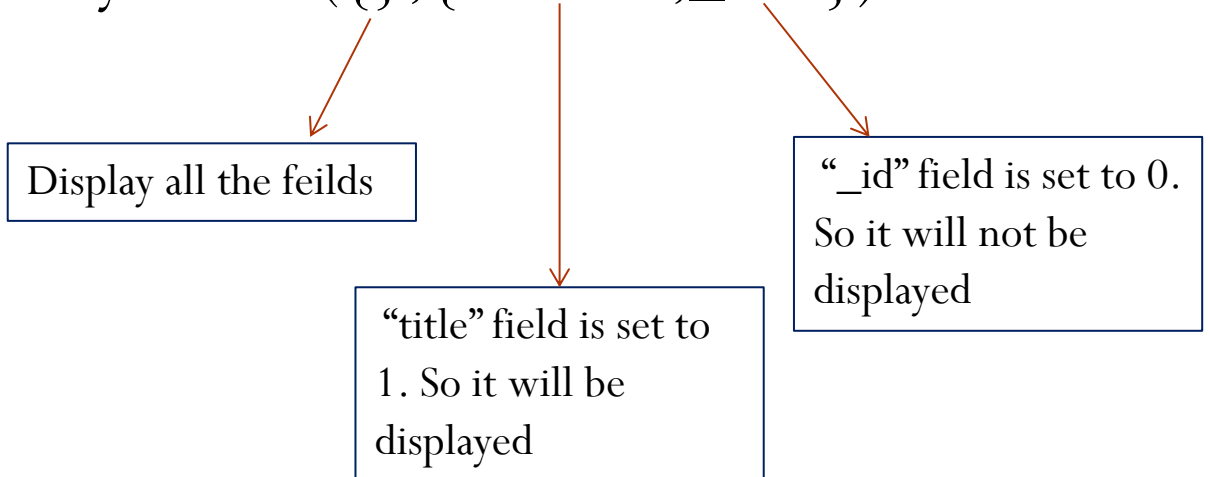

Projection

- **find()** method, then it displays all fields of a document. To limit this, you need to set a list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the fields.
- `>db.COLLECTION_NAME.find({}, {KEY:1})`
- `>db.mycol.find({}, {"title":1, _id:0})`
- Output:
 - `{"title":"MongoDB Overview"}`
 - `{"title":"NoSQL Overview"}`
 - `{"title":"Hbase Overview"}`
- Note: **_id** field is always displayed while executing **find()** method, if you don't want this field, then you need to set it as 0.

Projection

// Following example will display the title of the document while querying the document.

```
>db.mycol.find( {}, {"title":1,_id:0})
```



Display all the feilds

“title” field is set to 1. So it will be displayed

“_id” field is set to 0. So it will not be displayed

Sort

sort() method accepts a document containing a list of fields along with their sorting order.

To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

```
>db.COLLECTION_NAME.find().sort({KEY:1})
```

Eg.,

Following example will display the documents sorted by title in the descending order.

```
>db.mycol.find({}, {"title":1,_id:0}).sort({"title":-1})  
//By default sorting is ascending
```

Aggregate

- `>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)`
- `> db.mycol.aggregate([{ $group : { _id : "$by_user", num_tutorial : { $sum : 1 } } }])`
- **`select by_user, count(*) from mycol group by by_user.`**

References

- Textbook on Fundamentals of Database Systems, Elmasri Navathe, 7th Edition