Distributed database system

A distributed database is type of a database that has no limit to one system, it is spread over different sites, i.e, on multiple computers or over a network of computers.

A distributed database system is located on various sites that don't share physical components.

This may be required when a particular database needs to be accessed by various users globally.

It needs to be managed such that for the users it looks like one single database.

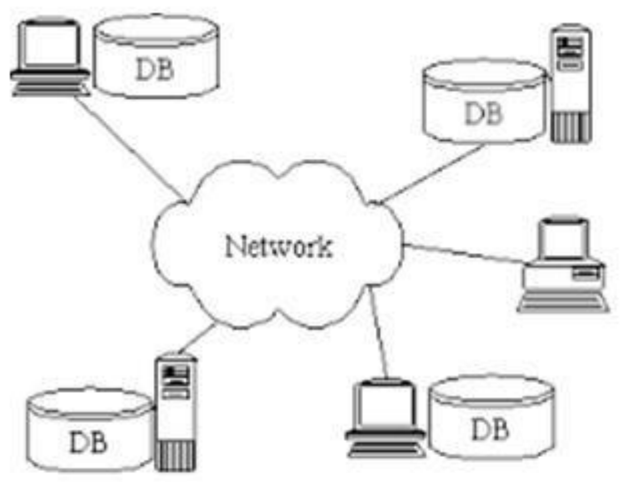


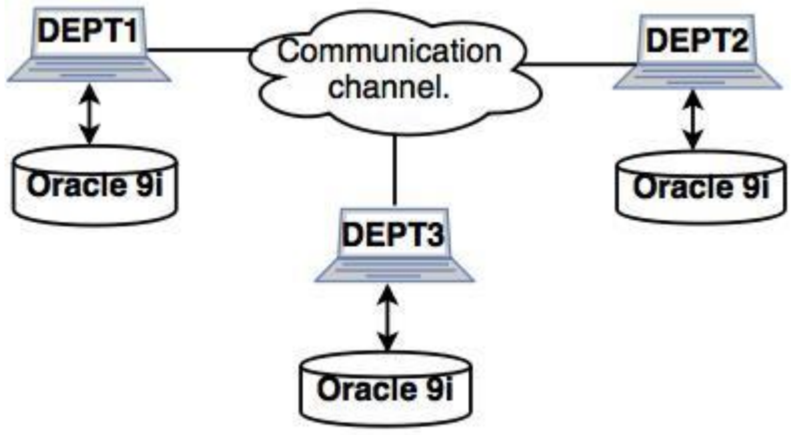
Figure. Distributed database System

Types of Distributed database

1. **Homogeneous Database:**

   In a homogeneous database, all different sites store database identically. The operating system, database management system, and the data structures used – all are the same at all sites. Hence, they're easy to manage.

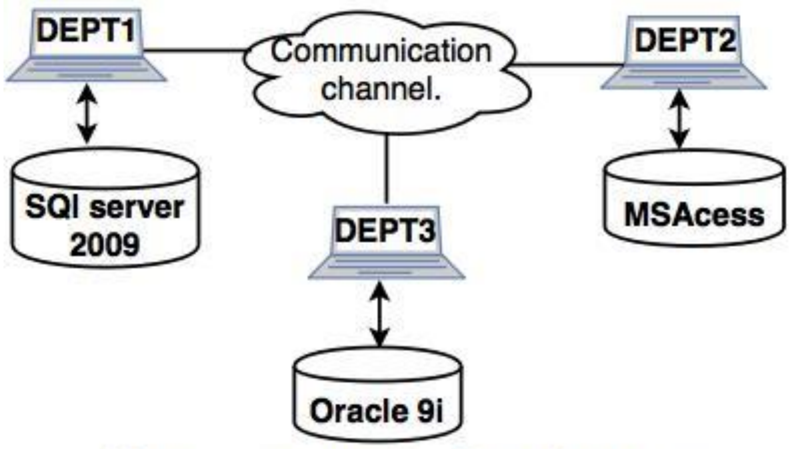
Example: Consider that we have three departments using Oracle-9i for DBMS. If some changes are made in one department then, it would update the other department also.



Homogeneous distributed system

2. **Heterogeneous Database:**

In a heterogeneous distributed database, different sites can use different schema and software that can lead to problems in query processing and transactions. Also, a particular site might be completely unaware of the other sites. Different computers may use a different operating system, different database application. They may even use different data models for the database. Hence, translations are required for different sites to communicate.



Heterogeneous distributed system

Applications of Distributed Database:

- It is used in Corporate Management Information System.
- It is used in multimedia applications.
- Used in Military's control system, Hotel chains etc.
- It is also used in manufacturing control system.

Distributed Data Storage :

There are 2 ways in which data can be stored on different sites. These are:

1. Replication –
In this approach, the entire relationship is stored redundantly at 2 or more sites.
If the entire database is available at all sites, it is a fully redundant database. Hence, in replication, systems maintain copies of data.
This is advantageous as it increases the availability of data at different sites. Also, now query requests can be processed in parallel.
However, it has certain disadvantages as well. Data needs to be constantly updated.

Any change made at one site needs to be recorded at every site that relation is stored or else it may lead to inconsistency.

This is a lot of overhead. Also, concurrency control becomes way more complex as concurrent access now needs to be checked over a number of sites.

2. Fragmentation –
In this approach, the relations are fragmented (i.e., they're divided into smaller parts) and each of the fragments is stored in different sites where they're required.
It must be made sure that the fragments are such that they can be used to reconstruct the original relation (i.e, there isn't any loss of data).
Fragmentation is advantageous as it doesn't create copies of data, consistency is not a problem.

Fragmentation of relations can be done in two ways:

Horizontal fragmentation – Splitting by rows –
The relation is fragmented into groups of tuples so that each tuple is assigned to at least one fragment.

- Horizontal fragmentation groups the tuples of a table in accordance to values of one or more fields. Horizontal fragmentation should also confirm to the rule of reconstructive Ness. Each horizontal fragment must have all columns of the original base table.

- For example, in the student schema, if the details of all students of Computer Science Course needs to be maintained at the School of Computer Science, then the designer will horizontally fragment the database as follows –

```
CREATE COMP_STD AS
  SELECT * FROM STUDENT
  WHERE COURSE = "Computer Science";
```

Vertical fragmentation – Splitting by columns –

The schema of the relation is divided into smaller schemas. Each fragment must contain a common candidate key so as to ensure a lossless join.

In vertical fragmentation, the fields or columns of a table are grouped into fragments. In order to maintain reconstructive Ness, each fragment should contain the primary key field(s) of the table. Vertical fragmentation can be used to enforce privacy of data.

For example, let us consider that a university database keeps records of all registered students in a Student table having the following schema.

STUDENT

Regd_No	Name	Course	Address	Semester	Fees	Marks

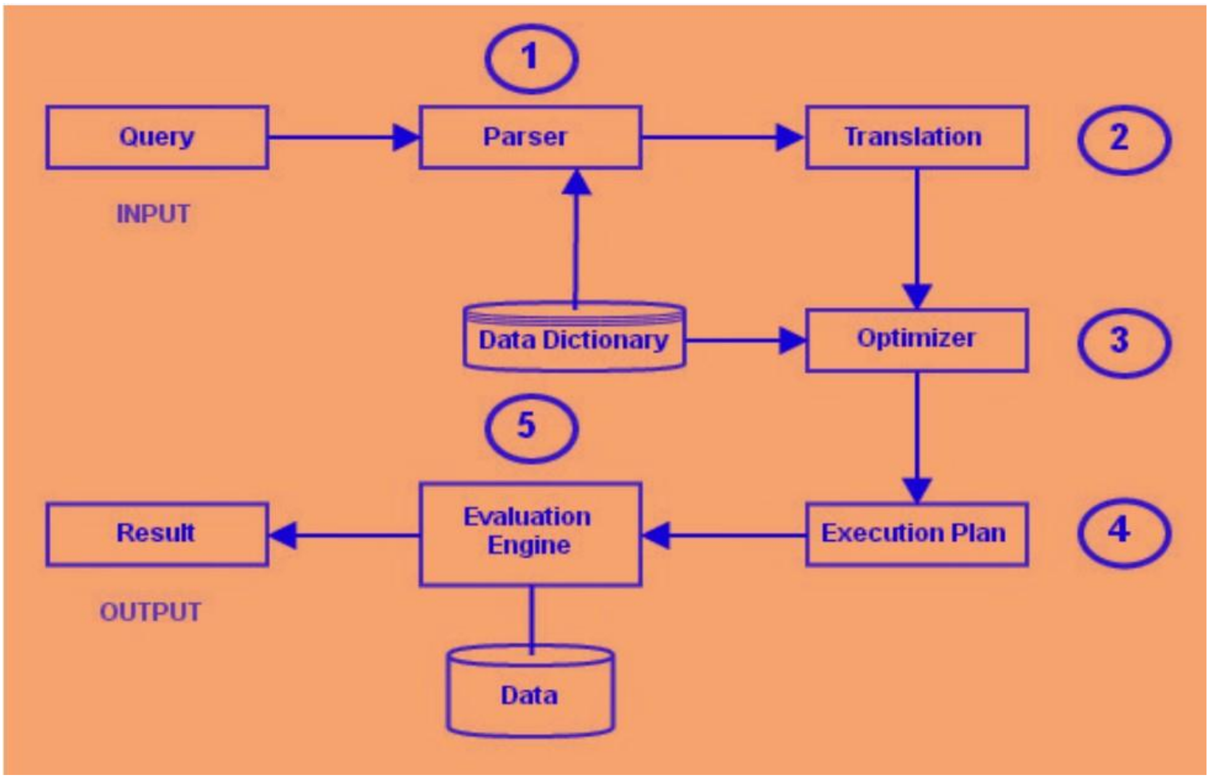
Now, the fees details are maintained in the accounts section. In this case, the designer will fragment the database as follows –

```
CREATE TABLE STD_FEES AS
  SELECT Regd_No, Fees
  FROM STUDENT;
```

In certain cases, an approach that is hybrid of fragmentation and replication is used.

Query Processing

Query processing is a set of all activities starting from query placement to displaying the results of the query. The steps are as shown in the following diagram –



Step 1: Parsing and translation

In this step of query processing in dbms the translator converts high level query provided by user to internal form where parser ***analyse this query and performs syntax, semantic and shared pool check*****.** It also verifies the name of the relation in the database, the tuple, and finally the required attribute value.

The parser constructs a tree of the query, known as '***parse-tree***'. And finally query is translated into relational algebra form. After this process all the usage of the views are replaced in the query.

Following checks are performed in parsing phase:

Syntax checking – In this step parser checks whether query is syntactically correct or not.

Example: *SELECT * Form Students*.

In the above query syntax check will fail as there is spelling error in from keyword.

Semantic checking – This step checks the meaningfulness of the sql query. It also verifies all table & column names from the dictionary and checks to see if you are authorised to see the data.

Example: Semantic check will fail if query refers to table in database which does not exist.

Shared Pool check – The next step in the parse operation is to see if the statement we are currently parsing has already in fact been processed by some other session. Every query is associated with hash code at the time of execution. This step checks whether there is existing hash code in shared pool. If there is a hash code already present DB will not perform optimisation and row source generation for this query

For example, our SQL query will be converted into a Relational Algebra equivalent as follows after this step:

$\pi_{student\_name}(\sigma_{percentage>90}\ (Students))$

Hard Parse Vs Soft Parse

When the DB finds the query already processed by some other session using shared pool check it skips next two steps of query processing in dbms i.e. **optimisation and row source generation** this is known as **Soft parsing**.If we cannot find the query in already processed pool, then we need do all of the steps, this is known as a **Hard Parsing**.

Step 2: Optimisation

This step of query processing in dbms analyses SQL queries and determines efficient execution mechanisms. Optimiser uses the statistical data stored as part of data dictionary. The statistical data are information about the size of the table, the length of records, the indexes created on the table, etc.

A query optimiser generates one or more query plans for each query, each of which may be a mechanism used to run a query. The most efficient query plan is selected and used to run the query.

Note: The database will never optimise DDL unless it has a DML component like a subqueries that require optimisation.

Row Source Generation

The next step in query optimisation is row source generation. The row source generator receives the optimal plan from the optimiser. The output of this step is execution plan for the SQL query.

The execution plan is a collection of row sources structured in the form of a tree. A *row source* is an iterative control structure. It processes a set of rows, one row at a time, in an iterated manner. A row source produces a row set.

Step 3: Evaluation

From the previous step we got many execution plans constructed through statistical data which all gives the same output, but they differ in terms of time and space consumption to execute the query. Hence, it becomes mandatory to choose one plan which obviously consumes less cost and is effective.

In this stage, we choose one execution plan from the several plans provided by previous step. This Execution plan accesses data from the database to give the final result.

For our SQL query, it can also be written as

$$\sigma_{percentage>90} (\pi_{student\_name} (Students))$$

The evaluation step picks up best execution plan for execution.

Step 4: Execution Engine

A query execution engine is responsible for generating the output of the given query. It takes the query execution plan chosen from previous step of query processing in dbms and executes it. Finally the output is displayed to the user.

Distributed Query Optimization

Distributed query optimization requires evaluation of a large number of query trees each of which produce the required results of a query. This is primarily due to the presence of large amount of replicated and fragmented data. Hence, the target is to find an optimal solution instead of the best solution.

The main issues for distributed query optimization are −

- Optimal utilization of resources in the distributed system.
- Query trading.
- Reduction of solution space of the query.

Semi join in Distributed Query processing:

The semi-join operation is used in distributed query processing to reduce the number of tuples in a table before transmitting it to another site. This reduction in the number of tuples reduces the number and the total size of the transmission that ultimately reducing the total cost of data transfer. Let’s say that we have two tables R1, R2 on Site S1, and S2. Now, we will forward the joining column of one table say R1 to the site where the other table say R2 is located. This column is joined with R2 at that site. The decision whether to reduce R1 or R2 can only be made after comparing the advantages of reducing R1 with that of reducing R2. Thus, semi-join is a well-organized solution to reduce the transfer of data in distributed query processing.

Example : Find the amount of data transferred to execute the same query given in the example using semi-join operation.

Consider the following table EMPLOYEE and DEPARTMENT.

Site1: **EMPLOYEE**

EID	NAME	SALARY	DID

EID-	10	bytes
SALARY-	20	bytes
DID-	10	bytes
Name-	20	bytes
Total	records-	1000

Record Size- 60 bytes

Site2: **DEPARTMENT**

DID DNAME

DID- 10 bytes
DName- 20 bytes
Total records- 50
Record Size- 30 bytes

Answer: The following strategy can be used to execute the query.

1. Select all (or Project) the attributes of the EMPLOYEE table at site 1 and then transfer them to site 3. For this, we will transfer NAME, DID(EMPLOYEE) and the size is 25 * 1000 = 25000 bytes.

2. Transfer the table DEPARTMENT to site 3 and join the projected attributes of EMPLOYEE with this table. The size of the DEPARTMENT table is 25 * 50 = 1250

Applying the above scheme, the amount of data transferred to execute the query will be 25000 + 1250 = 26250 bytes.

Parallel-Join

1. Parallel-join: split the pairs to be tested over several processors. Each processor computes part of the join, and then the results are assembled (merged).

2. Ideally, the overall work of computing join is partitioned evenly over all processors. If such a split is achieved without any overhead, a parallel join using *N* processors will take 1/*N* times as long as the same join would take on a single processor.

3. In practice, the speedup is less dramatic because-
    1. Overhead is incurred in partitioning the work among the processors.
    2. Overhead is incurred in collecting the results computed by each processor.
    3. If the split is not even, the final result cannot be obtained until the last processor has finished.
    4. The processors may compete for shared system resources, e.g., for $A \bowtie B$ (e.g., $deposit \bowtie customer$), if each processor uses its own partition of *A*, and the main memory cannot hold the entire *B*, the processors need to

synchronize the access of *B* so as to reduce the number of times that each block of *B* must be read in from disk.

4. A parallel hash algorithm to reduce memory contention.

   Choose a hash function whose range is $\{1, \ldots, N\}$ which allows us to assign each of the *N* processors to exactly one hash bucket. Since the final outer for-loop of the hash-join algorithm iterates over buckets, each processor can process the iteration that corresponds to its assigned bucket. Since no tuple is assigned to more than one bucket, so there is no contention for *B* tuples. Since each processor considers one pair of tuples at a time, the total main memory requirements of the parallel hash join algorithm are sufficiently low that contention for space in main memory is unlikely.

Concurrency Control In Distributed Database System

Concurrency controlling techniques ensure that multiple transactions are executed simultaneously while maintaining the ACID properties of the transactions and serializability in the schedules.

In this section, we will see how the above techniques are implemented in a distributed database system.

Distributed Two-phase Locking Algorithm

The basic principle of distributed two-phase locking is same as the basic two-phase locking protocol. However, in a distributed system there are sites designated as lock managers. A lock manager controls lock acquisition requests from transaction monitors. In order to enforce co-ordination between the lock managers in various sites, at least one site is given the authority to see all transactions and detect lock conflicts.

Depending upon the number of sites who can detect lock conflicts, distributed two-phase locking approaches can be of three types −

- **Centralized two-phase locking** − In this approach, one site is designated as the central lock manager. All the sites in the environment know the location of the central lock manager and obtain lock from it during transactions.
- **Primary copy two-phase locking** − In this approach, a number of sites are designated as lock control centers. Each of these sites has the responsibility of managing a defined set of locks. All the sites know which lock control center is responsible for managing lock of which data table/fragment item.
- **Distributed two-phase locking** − In this approach, there are a number of lock managers, where each lock manager controls locks of data items stored at its

local site. The location of the lock manager is based upon data distribution and replication.

Distributed Timestamp Concurrency Control

In a centralized system, timestamp of any transaction is determined by the physical clock reading. But, in a distributed system, any site's local physical/logical clock readings cannot be used as global timestamps, since they are not globally unique. So, a timestamp comprises of a combination of site ID and that site's clock reading.

For implementing timestamp ordering algorithms, each site has a scheduler that maintains a separate queue for each transaction manager. During transaction, a transaction manager sends a lock request to the site's scheduler. The scheduler puts the request to the corresponding queue in increasing timestamp order. Requests are processed from the front of the queues in the order of their timestamps, i.e. the oldest first.

Conflict Graphs

Another method is to create conflict graphs. For this transaction classes are defined. A transaction class contains two set of data items called read set and write set. A transaction belongs to a particular class if the transaction's read set is a subset of the class' read set and the transaction's write set is a subset of the class' write set. In the read phase, each transaction issues its read requests for the data items in its read set. In the write phase, each transaction issues its write requests.

A conflict graph is created for the classes to which active transactions belong. This contains a set of vertical, horizontal, and diagonal edges. A vertical edge connects two nodes within a class and denotes conflicts within the class. A horizontal edge connects two nodes across two classes and denotes a write-write conflict among different classes. A diagonal edge connects two nodes across two classes and denotes a write-read or a read-write conflict among two classes.

The conflict graphs are analyzed to ascertain whether two transactions within the same class or across two different classes can be run in parallel.

Recovery In Distributed Database System

In order to recuperate from database failure, database management systems resort to a number of recovery management techniques. In this chapter, we will study the different approaches for database recovery.

The typical strategies for database recovery are −

- In case of soft failures that result in inconsistency of database, recovery strategy includes transaction undo or rollback. However, sometimes,

transaction redo may also be adopted to recover to a consistent state of the transaction.
- In case of hard failures resulting in extensive damage to database, recovery strategies encompass restoring a past copy of the database from archival backup. A more current state of the database is obtained through redoing operations of committed transactions from transaction log.

Recovery from Power Failure

Power failure causes loss of information in the non-persistent memory. When power is restored, the operating system and the database management system restart. Recovery manager initiates recovery from the transaction logs.

In case of immediate update mode, the recovery manager takes the following actions −

- Transactions which are in active list and failed list are undone and written on the abort list.
- Transactions which are in before-commit list are redone.
- No action is taken for transactions in commit or abort lists.

In case of deferred update mode, the recovery manager takes the following actions −

- Transactions which are in the active list and failed list are written onto the abort list. No undo operations are required since the changes have not been written to the disk yet.
- Transactions which are in before-commit list are redone.
- No action is taken for transactions in commit or abort lists.

Recovery from Disk Failure

A disk failure or hard crash causes a total database loss. To recover from this hard crash, a new disk is prepared, then the operating system is restored, and finally the database is recovered using the database backup and transaction log. The recovery method is same for both immediate and deferred update modes.

The recovery manager takes the following actions −

- The transactions in the commit list and before-commit list are redone and written onto the commit list in the transaction log.
- The transactions in the active list and failed list are undone and written onto the abort list in the transaction log.

Transaction Recovery Using UNDO / REDO

Transaction recovery is done to eliminate the adverse effects of faulty transactions rather than to recover from a failure. Faulty transactions include all transactions that have changed the database into undesired state and the transactions that have used values written by the faulty transactions.

Transaction recovery in these cases is a two-step process −

- UNDO all faulty transactions and transactions that may be affected by the faulty transactions.
- REDO all transactions that are not faulty but have been undone due to the faulty transactions.

Steps for the UNDO operation are −

- If the faulty transaction has done INSERT, the recovery manager deletes the data item(s) inserted.
- If the faulty transaction has done DELETE, the recovery manager inserts the deleted data item(s) from the log.
- If the faulty transaction has done UPDATE, the recovery manager eliminates the value by writing the before-update value from the log.

Steps for the REDO operation are −

- If the transaction has done INSERT, the recovery manager generates an insert from the log.
- If the transaction has done DELETE, the recovery manager generates a delete from the log.
- If the transaction has done UPDATE, the recovery manager generates an update from the log.

Distributed Deadlock

Distributed deadlocks can occur when distributed transactions or concurrency control are utilized in distributed systems. It may be identified via a distributed technique like edge chasing or by creating a global wait-for graph (WFG) from local wait-for graphs at a deadlock detector. Phantom deadlocks are identified in a distributed system but do not exist due to internal system delays.

In a distributed system, deadlock cannot be prevented nor avoided because the system is too vast. As a result, only deadlock detection is possible. The following are required for distributed system deadlock detection techniques:

Detect deadlock in the distributed system

Various approaches to detect the deadlock in the distributed system are as follows:

1. Centralized Approach

Only one resource is responsible for detecting deadlock in the centralized method, and it is simple and easy to use. Still, the disadvantages include excessive workload on a single node and single-point failure (i.e., the entire system is dependent on one node, and if that node fails, the entire system crashes), making the system less reliable.

2. Hierarchical Approach

In a distributed system, it is the integration of both centralized and distributed approaches to deadlock detection. In this strategy, a single node handles a set of selected nodes or clusters of nodes that are in charge of deadlock detection.

3. Distributed Approach

In the distributed technique, various nodes work to detect deadlocks. There is no single point of failure as the workload is equally spread among all nodes. It also helps to increase the speed of deadlock detection.

Deadlock Handling in Distributed Systems

Transaction processing in a distributed database system is also distributed, i.e. the same transaction may be processing at more than one site. The two main deadlock handling concerns in a distributed database system that are not present in a centralized system are **transaction location** and **transaction control**. Once these concerns are addressed, deadlocks are handled through any of deadlock prevention, deadlock avoidance or deadlock detection and removal.

Transaction Location

Transactions in a distributed database system are processed in multiple sites and use data items in multiple sites. The amount of data processing is not uniformly distributed among these sites. The time period of processing also varies. Thus the same transaction may be active at some sites and inactive at others. When two conflicting transactions are located in a site, it may happen that one of them is in inactive state. This condition does not arise in a centralized system. This concern is called transaction location issue.

This concern may be addressed by Daisy Chain model. In this model, a transaction carries certain details when it moves from one site to another. Some of the details are the list of tables required, the list of sites required, the list of visited tables and sites, the list of tables and sites that are yet to be visited and the list of acquired locks with types. After a transaction terminates by either commit or abort, the information should be sent to all the concerned sites.

Transaction Control

Transaction control is concerned with designating and controlling the sites required for processing a transaction in a distributed database system. There are many options regarding the choice of where to process the transaction and how to designate the center of control, like −

- One server may be selected as the center of control.
- The center of control may travel from one server to another.

- The responsibility of controlling may be shared by a number of servers.

Distributed Deadlock Prevention

In distributed deadlock prevention approach, a transaction should acquire all the locks before starting to execute. This prevents deadlocks.

The site where the transaction enters is designated as the controlling site. The controlling site sends messages to the sites where the data items are located to lock the items. Then it waits for confirmation. When all the sites have confirmed that they have locked the data items, transaction starts. If any site or communication link fails, the transaction has to wait until they have been repaired.

Though the implementation is simple, this approach has some drawbacks −

- Pre-acquisition of locks requires a long time for communication delays. This increases the time required for transaction.
- In case of site or link failure, a transaction has to wait for a long time so that the sites recover. Meanwhile, in the running sites, the items are locked. This may prevent other transactions from executing.
- If the controlling site fails, it cannot communicate with the other sites. These sites continue to keep the locked data items in their locked state, thus resulting in blocking.

Distributed Deadlock Avoidance

As in centralized system, distributed deadlock avoidance handles deadlock prior to occurrence. Additionally, in distributed systems, transaction location and transaction control issues needs to be addressed. Due to the distributed nature of the transaction, the following conflicts may occur −

- Conflict between two transactions in the same site.
- Conflict between two transactions in different sites.

In case of conflict, one of the transactions may be aborted or allowed to wait as per distributed wait-die or distributed wound-wait algorithms.

Let us assume that there are two transactions, T1 and T2. T1 arrives at Site P and tries to lock a data item which is already locked by T2 at that site. Hence, there is a conflict at Site P. The algorithms are as follows –

- **Distributed Wound-Die**
    - If T1 is older than T2, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has either committed or aborted successfully at all sites.
    - If T1 is younger than T2, T1 is aborted. The concurrency control at Site P sends a message to all sites where T1 has visited to abort T1. The controlling site notifies the user when T1 has been successfully aborted in all the sites.

- **Distributed Wait-Wait**
    - If T1 is older than T2, T2 needs to be aborted. If T2 is active at Site P, Site P aborts and rolls back T2 and then broadcasts this message to other relevant sites. If T2 has left Site P but is active at Site Q, Site P broadcasts that T2 has been aborted; Site L then aborts and rolls back T2 and sends this message to all sites.
    - If T1 is younger than T1, T1 is allowed to wait. T1 can resume execution after Site P receives a message that T2 has completed processing.

Commit Protocols

In a local database system, for committing a transaction, the transaction manager has to only convey the decision to commit to the recovery manager. However, in a distributed system, the transaction manager should convey the decision to commit to all the servers in the various sites where the transaction is being executed and uniformly enforce the decision. When processing is complete at each site, it reaches the partially committed transaction state and waits for all other transactions to reach their partially committed states. When it receives the message that all the sites are ready to commit, it starts to commit. In a distributed system, either all sites commit or none of them does.

The different distributed commit protocols are −

- One-phase commit
- Two-phase commit
- Three-phase commit

Distributed One-phase Commit

Distributed one-phase commit is the simplest commit protocol. Let us consider that there is a controlling site and a number of slave sites where the transaction is being executed. The steps in distributed commit are −

- After each slave has locally completed its transaction, it sends a “DONE” message to the controlling site.
- The slaves wait for “Commit” or “Abort” message from the controlling site. This waiting time is called **window of vulnerability**.
- When the controlling site receives “DONE” message from each slave, it makes a decision to commit or abort. This is called the commit point. Then, it sends this message to all the slaves.
- On receiving this message, a slave either commits or aborts and then sends an acknowledgement message to the controlling site.

Distributed Two-phase Commit

Distributed two-phase commit reduces the vulnerability of one-phase commit protocols. The steps performed in the two phases are as follows −

Phase 1: Prepare Phase

- After each slave has locally completed its transaction, it sends a "DONE" message to the controlling site. When the controlling site has received "DONE" message from all slaves, it sends a "Prepare" message to the slaves.
- The slaves vote on whether they still want to commit or not. If a slave wants to commit, it sends a "Ready" message.
- A slave that does not want to commit sends a "Not Ready" message. This may happen when the slave has conflicting concurrent transactions or there is a timeout.

Phase 2: Commit/Abort Phase

- After the controlling site has received "Ready" message from all the slaves −
    - The controlling site sends a "Global Commit" message to the slaves.
    - The slaves apply the transaction and send a "Commit ACK" message to the controlling site.
    - When the controlling site receives "Commit ACK" message from all the slaves, it considers the transaction as committed.
- After the controlling site has received the first "Not Ready" message from any slave −
    - The controlling site sends a "Global Abort" message to the slaves.
    - The slaves abort the transaction and send a "Abort ACK" message to the controlling site.
    - When the controlling site receives "Abort ACK" message from all the slaves, it considers the transaction as aborted.

Distributed Three-phase Commit

The steps in distributed three-phase commit are as follows −

Phase 1: Prepare Phase

The steps are same as in distributed two-phase commit.

Phase 2: Prepare to Commit Phase

- The controlling site issues an "Enter Prepared State" broadcast message.
- The slave sites vote "OK" in response.

Phase 3: Commit / Abort Phase

The steps are same as two-phase commit except that "Commit ACK"/"Abort ACK" message is not required.