UNIT IV

THE MANAGEMENT OF DISTRIBUTED TRANSACTIONS

INTRODUCTION

Distributed database systems are systems that have their data distributed and replicated over several locations, not like the centralized database system, where one copy of the data is stored. Data may be replicated over a network using horizontal and vertical fragmentation similar to projection and selection operations in Structured Query Language (SQL). In other words, Distributed database as a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database management system (DDBMS) is then defined as the software system that permits the management of the distributed database and makes the distribution transparent to the users. Sometimes "distributed database system" (DDBS) is used to refer jointly to the distributed database and the distributed DBMS. Both types of database share the same problems of access control and transaction management, such as user concurrent access control and deadlock detection and resolution. On the other hand, however, DDBS must also cope with different problems. Access control and transaction management in DDBS require different rules to monitor data retrieval and update to distributed and replicated databases. Oracle, as a leading Database Management Systems (DBMS) employs the two-phase commit technique to maintain a consistent state for the databases. The objective of this paper is to explain transaction management in DDBMS and how Oracle implements this technique. To assist in understanding this process, an example is given in the last section. It is hoped that this understanding will encourage organizations to use and academics to discuss DDBS and to successfully capitalize on this feature of Oracle. It provide discussions on the fundamentals of transaction management, two-phase commit, Oracle's implementation of the two phase commit, and, finally, an example on how the two phases commit works.

Transaction Management deals with the problems of keeping the database in a consistent state even when concurrent accesses and failures occurs

TRANSACTION:

A transaction consists of a series of operations performed on a database. The important issue in transaction management is that if a database was in a consistent state prior to the initiation of a transaction, then the database should return to a consistent state after the transaction is completed. This should be done irrespective of the fact that transactions were

successfully executed simultaneously or there were failures during the execution [2]. A transaction is a sequence of operations that takes the database from a consistent state to another consistent state. It represents a complete and correct computation.

Two types of transactions are allowed in our environment: query transactions and update transactions. Query transactions consist only of read operations that access data objects and return their values to the user. Thus, query transactions do not modify the database state. Two transactions conflict if the read-set of one transaction intersects with the write-set of the other transaction. During the voting process, Update transactions consist of both read and write operations. Transactions have their time-stamps constructed by adding 1 to the greater of either the current time or the highest time-stamp of their base variables. Thus; a transaction is a unit of consistency and reliability. Each transaction has to terminate. The outcome of the termination depends on the success or failure of the transaction. When a transaction starts executing, it may terminate with one of two possibilities:

1. The transaction **aborts** if a failure occurred during its execution
2. The transaction **commits** if it was completed successfully example of a transaction that aborts during process 2 (P2). On the other hand, an example of a transaction that commits, since all of its processes are successfully completed [3].

PROPERTIES OF TRANSACTIONS

A TRANSACTION HAS FOUR PROPERTIES THAT LEAD TO THE CONSISTENCY AND RELIABILITY OF A DISTRIBUTEDDATA BASE. THESE ARE ATOMICITY, CONSISTENCY, ISOLATION AND DURABILITY.

ATOMICITY

This refers to the fact that a transaction is treated as a unit of operation. Consequently, it dictates that either all the actions related to a transaction are completed or none of them is carried out. For example, in the case of a crash, the system should complete the remainder of the transaction, or it will undo all the actions pertaining to this transaction. The recovery of the transaction is split into two types corresponding to the two types of failures: the transaction recovery, which is due to the system terminating one of the transactions because of deadlock handling; and the crash recovery, which is done after a system crash or a hardware failure.

CONSISTENCY

Referring to its correctness, this property deals with maintaining consistent data in a database system. Consistency falls under the subject of concurrency control. For example, “dirty data” is data that has been modified by a transaction that has not yet committed. Thus, the job of concurrency control is to be able to disallow transactions from reading or updating ‘dirty data’.

ISOLATION

According to this property, each transaction should see a consistent database at all times. Consequently, no other transaction can read or modify data that is being modified by another transaction. If this property is not maintained, one of two things could happen to the data base. a. Lost Updates: this occurs when transaction (T2) updates the same data being modified by the transaction (T1) in such a manner that T2 reads the value prior to the writing of T1 thus creating the problem of loosing this update.

b. Cascading Aborts: this problem occurs when the first transaction (T1) aborts, then the transactions that had read or modified data that has been used by T1 will also abort.

Durability

This property ensures that once a transaction commits, its results are permanent and cannot be erased from the database. This means that whatever happens after the COMMIT of a transaction, whether it is a system crash or aborts of other transactions, the results already committed are not modified or undone.

1. DISTRIBUTED TRANSACTION MANAGEMENT

This section provides general background on the atomic commit problem and protocols.

1.1. PROBLEM DEFINITION

A distributed transaction is composed of several sub-transactions, each running on a different site. The database manager at each site can unilaterally decide to abort the local sub- transaction, in which case the entire transaction must be aborted. If all the participating sites agree to commit their sub-transaction (vote Yes on the transaction) and no failures occur, the transaction should be committed. I assume that the local database server at each site can atomically execute the sub-transaction once it has agreed to commit it. In order to ensure that all the sub-transactions are consistently committed or aborted,

the sites run an atomic commitment protocol such as two phases commit. The requirements of atomic commitment [7]) are as follows:

AC1: Uniform Agreement: All the sites that reach a decision reach the same one.

AC2: A site cannot reverse its decision after it has reached one.

AC3: Validity: The commit decision can be reached only if all sites voted Yes.

AC4: Non-triviality: If there are no failures and all sites voted Yes, then the decision will be to commit.

AC5: Termination: At any point in the execution of the protocol, if all existing failures are repaired and no new failures occur for sufficiently long, then all sites will even- tually reach a decision.

THE PREPARE AND COMMIT

PHASESPREPARE PHASE

The first phase in committing a distributed transaction is the prepare phase in which the commit of the transaction is not actually carried out. Instead, all nodes referenced in a distributed transaction (except one, known as the commit point site) are told to prepare (to commit). By preparing, a node records enough information so that it can subsequently either commit or abort the transaction (in which case, a rollback will be performed), regardless of intervening failures. Prepare phase; the global coordinator (initiating node) asks participants to prepare (to promise to commit or rollback the transaction, even if there is a failure).

Commit phase; if all participants respond to the coordinator that they are prepared, the coordinator asks all nodes to commit the transaction. If any participants cannot prepare, the coordinator asks all nodes to roll back the transaction.

When a node responds to its requestor that it has prepared, the prepared node has made a promise to be able to either commit or roll back the transaction later and not to make a unilateral decision on whether to commit or roll back the transaction [6].

Queries that start after a node has prepared cannot access the associated locked data until all phases are complete (an insignificant amount of time unless a failure occurs). When a node is told to prepare, it can respond with one of three responses:

PREPARE PHASE ACTIONS BY NODES

To complete the prepare phase, each node performs the following actions:

- The node requests its descendants (nodes subsequently referenced) to prepare.
- The node checks to see if the transaction changes data on that node or any of its descendants. If there is no change, the node skips the next steps and replies with a read-only message.
- The node allocates all resources it needs to commit the transaction if data is changed.
- The node flushes any entries corresponding to changes made by that transaction to its local redo log.
- The node guarantees that locks held for that transaction are able to survive a failure.
- The node responds to the node that referenced it in the distributed transaction with a prepared message *or*, if its prepare or the prepare of one of its descendents was unsuccessful, with an abort message.

These actions guarantee that the transaction can subsequently commit or roll back on that node. The prepared nodes then wait until a COMMIT or ROLLBACK is sent. Once the node(s) are prepared, the transaction is said to be *in-doubt*. Prepared Data on the node has been modified by a statement in the distributed transaction, and the node has successfully prepared. Read-only No data on the node has been, or can be, modified (only queried), so no prepare is necessary. Abort The node cannot successfully prepare.

READ-ONLY RESPONSE

When a node is asked to prepare and the SQL statements affecting the database do not change that node's data, the node responds to the node that referenced it with a read-only message. These nodes do not participate in the second phase (the commit phase).

UNSUCCESSFUL PREPARE

When a node cannot successfully prepare, it performs the following actions:

- That node releases any resources currently held by the transaction and rolls back the local portion of the transaction.
- The node responds to the node that referenced it in the distributed transaction with an *abort message*.

These actions then propagate to the other nodes involved in the distributed transaction to roll back the transaction and guarantee the integrity of the data in the global database.

Again, this enforces the primary rule of a distributed transaction. All nodes involved in the transaction either all commit or all roll back the transaction at the same logical time.

COMMIT PHASE

The second phase in committing a distributed transaction is the commit phase. Before this phase occurs, *all* nodes referenced in the distributed transaction have guaranteed that they have the necessary resources to commit the transaction. That is, they are all prepared. Therefore, the commit phase consists of the following steps [6]:

1. The global coordinator sends a message to all nodes telling them to commit the transaction.

2. At each node, Oracle commits the local portion of the distributed transaction (releasing locks) and records an additional redo entry in the local redo log, indicating that the transaction has committed. When the commit phase is complete, the data on all nodes of the distributed system are consistent with one another. A variety of failure cases, caused by network or system failures, are possible during both the prepare phase and the commit phase.

1.2. TWO PHASE COMMIT

The simplest and most renowned atomic commitment protocol (ACP) is two phases commit [7]. Several variations of 2PC have been suggested (e.g., presume abort and presume commit); the simplest version is centralized; one of the sites is designated as the coordinator. The coordinator sends a transaction (or request to prepare to commit) to all the participants. Each site answers by a Yes (“ready to commit'') or by a No (“abort'') message. If any site votes No, all the sites abort.

The 2-phase commit (2PC) protocol is a distributed algorithm to ensure the consistent termination of a transaction in a distributed environment. Thus, via 2PC a unanimous decision is reached and enforced among multiple participating servers whether to commit or abort a given transaction, thereby guaranteeing atomicity. The protocol proceeds in two phases, namely the prepare and the commit phase, which explains the protocol‘s name. The protocol is executed by a coordinator process, while the participating servers are called participants. When the transaction‘s initiator issues a request to commit the transaction, the coordinator starts the first phase of the 2PC protocol by querying—via prepare messages—all participants whether to abort or to commit the transaction The master

initiates the first phase of the protocol by sending *PREPARE* (to commit) messages in parallel to all the cohorts. Each cohort that is ready to commit first force-writes a *prepare* log record to its local stable storage and then sends a *YES* vote to the master. At this stage, the cohort has entered a *prepared* state wherein it cannot unilaterally commit or abort the transaction but has to wait for the final decision from the master. On the other hand, each cohort that decides to abort force-writes an *abort log* record and sends a NO vote to the master. Since a NO vote acts like a veto, the cohort is permitted to unilaterally abort the transaction without waiting for a response from the master. After the master receives the votes from all the cohorts, it initiates the second phase of the protocol. If all the votes are *YES,* it moves to a *committing* state by force writing a *commit* log record and sending *COMMIT* messages to all the cohorts. Each cohort after receiving a *COMMIT* message moves to the committing state, force-writes a *commit log* record, and sends an *ACK* message to the master. If the master receives even one *NO* vote, it moves to the aborting state by force-writing an *abort log* record and sends *ABORT* messages to those cohorts that are in the prepared state. These cohorts, after receiving the *ABORT* message, move to the *aborting* state, force write an *abort log* record and send an *ACK* message to the master [8].

Finally, the master, after receiving acknowledgements from all the prepared cohorts, writes an *end* log record and then ―forgets‖ the transaction. The 2PC may be carried out with one of the following methods: Centralized 2PC, Linear 2PC, and Distributed 2PC, [3].

1.3. THE CENTRALIZED TWO-PHASE COMMIT PROTOCOL

In the Centralized 2PC shown in Figure 3, communication is done through the coordinator‘s process only, and thus no communication between subordinates is allowed. The coordinator is responsible for transmitting the PREPARE message to the subordinates, and, when the votes of all the subordinates are received and evaluated, the coordinator decides on the course of action: either abort or COMMIT. This method has two phases:

1. First Phase: In this phase, when a user wants to COMMIT a transaction, the coordinator issues a PREPARE message to all the subordinates, (Mohan et al., 1986). When a subordinate receives the PREPARE message, it writes a PREPARE log and, if that subordinate is willing to COMMIT, sends a YES VOTE, and enters the PREPARED state; or, it writes an abort record and, if that subordinate is not willing to COMMIT, sends a NO VOTE. A subordinate sending a NO VOTE doesn‘t need to enter a PREPARED state since it knows that

the coordinator will issue an abort. In this case, the NO VOTE acts like a veto in the sense that only one NO VOTE is needed to abort the transaction. The following two rules apply to the coordinator's decision, *3].

a. If even one participant votes to abort the transaction, the coordinator has to reach a global abort decision.

b. If all the participants vote to COMMIT, the coordinator has to reach a global COMMIT decision.

2. Second Phase: After the coordinator reaches a vote, it has to relay that vote to the subordinates. If the decision is COMMIT, then the coordinator moves into the committing state and sends a COMMIT message to all the subordinates informing them of the COMMIT. When the subordinates receive the COMMIT message, they, in turn, move to the committing state and send an acknowledge (ACK) message to the coordinator. When the coordinator receives the ACK messages, it ends the transaction. If, on the other hand, the coordinator reaches an ABORT decision, it sends an ABORT message to all the subordinates. Here, the coordinator doesn't need to send an ABORT message to the subordinate(s) that gave a NO VOTE.

1.4. THE LINEAR TWO-PHASE COMMIT PROTOCOL

In the linear 2PC, as depicted in Figure 4, subordinates can communicate with each other. The sites are labeled 1 to N, where the coordinator is numbered as site 1. Accordingly, the propagation of the PREPARE message is done serially. As such, the time required to complete the transaction is longer than centralized or distributed methods. Finally, node N is the one that issues the Global COMMIT. The two phases are discussed below, [3]:

First Phase: The coordinator sends a PREPARE message to participant 2. If participant 2 is not willing to COMMIT, then it sends a VOTE ABORT (VA) to participant 3 and the transaction is aborted at this point. If participant 2, on the other hand, is willing to commit, it sends a VOTE COMMIT (VC) to participant 3 and enters a READY state. In turn, participant 3 sends its vote till node N is reached and issues its vote.

Second Phase: Node N issues either a GLOBAL ABORT (GA) or a GLOBAL COMMIT (GC) and sends it to node N-1. Subsequently, node N-1 will enter an ABORT or COMMIT state. In turn, node N-1 will send the GA or GC to node N-2, until the final vote to commit or abort reaches the coordinator, node .

1.5. THE DISTRIBUTED TWO-PHASE COMMIT PROTOCOL

In the distributed 2PC, all the nodes communicate with each other. According to this protocol, as Figure 5 shows, the second phase is not needed as in other 2PC methods. Moreover, each node must have a list of all the participating nodes in order to know that each node has sent in its vote. The distributed 2PC starts when the coordinator sends a PREPARE message to all the participating nodes. When each participant gets the PREPARE message, it sends its vote to all the other participants. As such, each node maintains a complete list of the participants in every transaction,[3]. Each participant has to wait and receive the vote from all other participants. When a node receives all the votes from all the participants, it can decide directly on COMMIT or abort. There is no need to start the second phase, since the coordinator does not have to consolidate all the votes in order to arrive at the final decision.

The coordinator collects all the responses and informs all the sites of the decision. In absence of failures, this protocol preserves atomicity. Between the two phases, each site blocks, i.e., keeps the local database locked, waiting for the final word from the coordinator. If a site fails before its vote reaches the coordinator, it is usually assumed that it had voted No. If the coordinator fails in the first phase, all the sites remain blocked indefinitely, unable to resolve the last transaction. The centralized version of 2PC is depicted in Fig. 1. Commit protocols may also be described using state diagrams [7]. The state diagram for 2PC is shown in Fig. 1. The circles denote states; final states are double- circled. The arcs represent state transitions, and the action taken (e.g., message sent) by the site is indicated next to each arc. In this protocol, each site (either coordinator or participant) can be in one of four possible states: q: initial state; A site is in the initial state until it decides whether to unilaterally abort or to agree to commit the transaction. w: wait state; In this state the coordinator waits for votes from all of the participants, and each participant waits for the final work from the coordinator. This is the “uncertainty period'' for each site, when it does not know whether the transaction will be committed or not. c: commit state; The site knows that a decision to commit was made.

a: abort state; The site knows that a decision to abort was made. The states of a commit protocol may be classified along two orthogonal lines. In the first dimension, the states are divided into two disjoint subsets: The committable states and the non-committable states. A

site is in a committable state only if it knows that all the sites have agreed to proceed with the trans-action. The rest of the states are non-committable. The only committable state in 2PC is the commit state. The second dimension distinguishes between final and non-final states. The final states are the ones in which a decision has been made and no more state transitions are possible. The final states in 2PC are commit and abort, [7].

THE COMMIT POINT SITE

The job of the commit point site is to initiate a commit or roll back as instructed by the global coordinator. The system administrator always designates one node to be the *commit point site* in the session tree by assigning all nodes commits point strength. The node selected as commit point site should be that node that stores the most critical data (the data most widely used) The commit point site is distinct from all other nodes involved in a distributed transaction with respect to the following two issues:

1. The commit point site never enters the prepared state. This is potentially advantageous because if the commit point site stores the most critical data, this data never remains in-doubt, even if a failure situation occurs. (In failure situations, failed nodes remain in a prepared state, holding necessary locks on data until in-doubt transactions are resolved.)
2. In effect, the outcome of a distributed transaction at the commit point site determines whether the transaction at all nodes is committed or rolled back. The global coordinator ensures that all nodes complete the transaction the same way that the commit point site does. A distributed transaction is considered to be committed once all nodes are prepared and the transaction has been committed at the commit point site (even though some participating nodes may still be only in the prepared state and the transaction not yet actually committed). The commit point site's redo log is updated as soon as the distributed transaction is committed at that node, [6]. Likewise, a distributed transaction is considered *not* committed if it has not been committed at the commit point site.

FAILURES THAT INTERRUPT TWO-PHASE COMMIT

The user program that commits a distributed transaction is informed of a problem by one of the following error messages:

ORA-02050: transaction ID rolled back,

some remote dbs may be in-doubt

ORA-02051: transaction ID committed,

some remote dbs may be in-doubt

ORA-02054: transaction ID in-doubt

A robust application should save information about a transaction if it receives any of the above errors. This information can be used later if manual distributed transaction recovery is desired.

Note: The failure cases that prompt these error messages are beyond the scope of this book and are unnecessary to administer the system. No action is required by the administrator of any node that has one or more indoubt distributed transactions due to a network or system failure. The automatic recovery features of Oracle transparently complete any in-doubt transaction so that the same outcome occurs on all nodes of a session tree (that is, all commit or all roll back) once the network or system failure is resolved. However, in extended outages, the administrator may wish to force the commit or rollback of a transaction to release any locked data. Applications must account for such possibilities.

FAILURES THAT PREVENT DATA ACCESS

When a user issues a SQL statement, Oracle attempts to lock the required resources to successfully execute the statement. However, if the requested data is currently being held by statements of other uncommitted transactions and continues to remain locked for an excessive amount of time, a time-out occurs. Consider the following two scenarios, [6].

TRANSACTION TIME-OUT

A DML SQL statement that requires locks on a remote database may be blocked from doing so if another transaction (distributed or non-distributed) currently own locks on the requested data. If these locks continue to block the requesting SQL statement, a time-out occurs, the statement is rolled back, and the following error message is returned to the user:

ORA-02049: time-out: distributed transaction waiting for lock, Because no data has been modified, no actions are necessary as a result of the timeout. Applications should proceed as if a deadlock has been encountered. The user who executed the statement can try to re-execute the statement later. If the lock persists, the user should contact an administrator to report the problem. The timeout interval in the above situation can be controlled with the initialization parameter DISTRIBUTED_LOCK_TIMEOUT. This interval is in seconds. For

example, to set the time-out interval for an instance to 30 seconds, include the following line in the associated parameter file:

DISTRIBUTED_LOCK_TIMEOUT=30

With the above time-out interval, the time-out errors discussed in the previous section occur if a transaction cannot proceed after 30 seconds of waiting for unavailable resources.

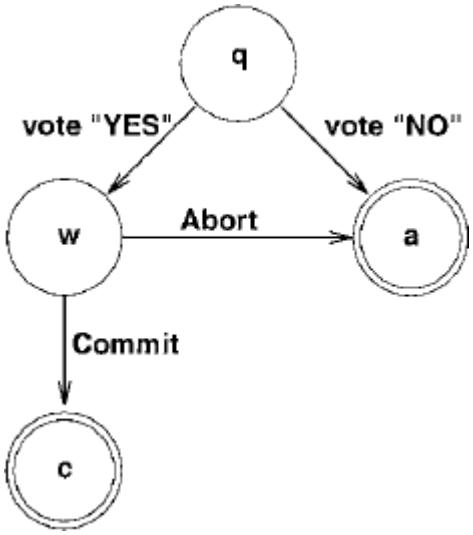
1.6. QUORUMS

In order to reduce the extent of blocking in replication and atomic commit protocols, majority votes or quorums are often used. A quorum system is a generalization of the majority concept. Enhanced three phase commit (E3PC), like Skeen's quorum-based three phase commit protocol , uses a quorum system to decide when a group of connected sites may resolve the transaction. To enable maximum flexibility the quorum system may be elected in a variety of ways (e.g., weighted voting). The quorum system is static; it does not change in the course of the protocol. The predicate Q(S) is true for a given subset S of the sites iff S is a quorum. The requirement from this predicate is that for any two sets of sites S and S' such that $S \cap S' = \Phi$, at most one of Q(S) and Q(S') holds, i.e., every pair of quorums intersect. For example, in the simple majority quorum system Q(S) is true iff $|S| > n/2$, where n is the total number of sites running the protocol. Numerous quorum systems that fulfill these criteria were suggested. An analysis of the availability of different quorum systems may be found in, [9].

For further flexibility, it is possible to set different quorums for commit and abort (this idea was presented in [7]). In this case, a commit quorum of connected sites is required in order to commit a transaction, and an abort quorum is required to abort. For example, to increase the probability of commit in the system, one can assign smaller quorums for commit and larger ones for abort. In this case, the quorum system consists of two predicates: $Q_C(G)$ is true for a given group of sites G iff G is a commit quorum, and $Q_A(G)$ is true iff G is an abort quorum. The requirement from these predicates is that for any two groups of sites G and G' such that $G \& G' = \Phi$, at most one of $Q_C(G)$ and $Q_A(G')$ holds, i.e., every commit quorum intersects every abort quorum.

1.7. THE EXTENT OF BLOCKING IN COMMIT PROTOCOLS

The 2PC protocol is an example of a blocking protocol,[7]: operational sites sometimes wait on the recovery of failed sites.



Locks must be held in the database while the transaction is blocked. Even though blocking preserves consistency, it is highly undesirable because the locks acquired by the blocked transaction cannot be relinquished, render- ing the data inaccessible by other requests. Consequently, the availability of data stored in reliable sites can be limited by the availability of the weakest component in the distributed system; [7] proved that there exists no non-blocking protocol resilient to network partitioning. When a partition occurs, the best protocols allow no more than one group of sites to continue while the remaining groups block. Skeen suggested the quorum-based three phase commit protocol, which maintains consistency in spite of network partitions. This protocol is blocking in case of partitions; it is possible for an operation site to be blocked until a failure is mended. In case of failures, the algorithm uses a quorum (or majority)-based recovery procedure that allows a quorum to resolve the transaction. If failures cascade, however, a quorum of sites can become connected and still remain blocked.

Since completely non-blocking recovery is impossible to achieve, further research in this area concentrated on minimizing the number of blocked sites when partitions occur. define optimal termination protocols (recovery procedures) in terms of the average number of sites that are blocked when a partition occurs. The average is over all the possible partitions and all the possible states in the protocol in which the partitions occur. The analysis deals only with states in the basic commit protocol and ignore the possibility for cascading failures (failures that occur during the recovery procedure). It is proved that any ACP with optimal recovery procedures takes at least three phases and that the quorum-based recovery procedures are optimal. I construct an ACP that always allows a connected majority to

proceed, regardless of past failures. To our knowledge, no other ACP with this feature was suggested. The ACP suggested in uses a reliable replication service as a building block and is mainly suitable for replicated database systems. In this paper, I present a novel commitment protocol, enhanced three phase commit, which always allows a connected majority to resolve the transaction (if it remains connected for sufficiently long). E3PC does not require complex building blocks, and is more adequate for partially replicated or non replicated distributed database systems; it is based on the quorum-based three phase commit, [7].

AN EXAMPLE OF A DISTRIBUTED DATABASE SYSTEM

Fig. 2 illustrates the steps homogenous distributed database performs in order to PREPARE, Select the COMMIT Point Site, and COMMIT. The example in the figure depicts a company that has several branches located in different cities numbered A to G. Each site has to have access to most of the data in the company in order to check on the status of purchase orders, material acquisition, and several other issues. Since new projects are awarded and older projects are completed, project sites tend to change locations. Also, depending on the size and duration of a project, different COMMIT point strength can be assigned and thus, in the same area, different COMMIT point sites can be chosen, for a given location, over a period of time. In this example, City E is the head office and thus posses the highest COMMIT point strength. The other sites are assigned the COMMIT point strength based on the rupee volume of the project. Higher monetary value for a project requires more resource allocation, and as such, will lead to more transactions executed against the data for that project. Since the amount of data involved is large, each site will have the portion of the database pertaining to its operations replicated and stored on a local server. Any transaction will at least affect the database at the head office and one of the sites. If, for example, a material rate, description of an item, accomplished progress, or purchase order is entered, a transaction is initiated that will affect the database at the head office and the database at the concerned site, [3].

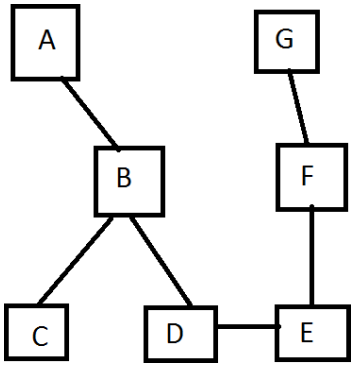


Fig.2-Distributed Database System over several node

Additional modifications, such as those involving employee transfer or equipment transfer from one site to another, will affect two or more sites. The following discussion explains the steps that entail in processing a distributed transaction: An employee is to be transferred from City F to City B. The transaction is initiated by City E by a personnel employee. The affected sites need to participate in the transaction. The processes that transfer one employee from one site to another should be grouped under one transaction so that either all or none of the processes are carried out. An explanation of these steps follows, [3]:

1. Since City E is initiating the transaction, it becomes the root of the session tree, i.e. the global coordinator. Since City 1 updates data in City F and City B, it becomes a client. Since City E updates data on City G and City B, the two nodes become database servers.

2. When the application issues the COMMIT statement, the two-phase commit is started.

3. The global coordinator determines the COMMIT point site.

4. The global coordinator issues the PREPARE statement to all nodes except the COMMIT point site. If any of the nodes cannot PREPARE, the transaction is aborted; otherwise, a PREPARED message is sent to the node that referenced it.

5. The global coordinator instructs the COMMIT point site to COMMIT. The COMMIT point site commits the transaction locally and records the transaction in its local redo log.

6. The COMMIT point site informs the global coordinator that it has committed and the global coordinator informs the other nodes by sending the COMMIT message.

7. When all the transactions have committed, the global coordinator informs the COMMIT point site to —forget|| about the transaction. The COMMIT point site, after —forgetting|| about the transaction, informs the global coordinator, and the global coordinator, in turn, —forgets|| about the transaction.

2.1. FAILURES IN DISTRIBUTED DBS

Several types of failures, [2] may occur in distributed database systems:

Transaction Failures: When a transaction fails, it aborts. Thereby, the database must be restored to the state it was in before the transaction started. Transactions may fail for several reasons. Some failures may be due to deadlock situations or concurrency control algorithms. Site Failures; Site failures are usually due to software or hardware failures. These failures result in the loss of the main memory contents. In distributed database, site failures are of two types:

1. **Total Failure:** where all the sites of a distributed system fail,
2. **Partial Failure:** where only some of the sites of a distributed system fail.

Media Failures: Such failures refer to the failure of secondary storage devices. The failureitself may be due to head crashes, or controller failure. In these cases, the media failuresresult in the inaccessibility of part or the entire database stored on such secondary storage. **Communication Failures:** Communication failures, as the name implies, are failures in thecommunication system between two or more sites. This will lead to network partitioningwhere each site, or several sites grouped together, operates independently. As such,messages from one site won't reach the other sites and will therefore be lost. The reliabilityprotocols then utilize a timeout mechanism in order to detect undelivered messages. Amessage is undelivered if the sender doesn't receive an acknowledgment. The failure of acommunication network to deliver messages is known as performance failure, [1].