UNIT - IV

Distributed DBMS Reliability: Reliability concepts and measures, fault-tolerance in distributed systems, failures in Distributed DBMS, local & distributed reliability protocols, site failures and network partitioning.

Parallel Database Systems: Parallel database system architectures, parallel data placement, parallel query processing, load balancing, database clusters.

Distributed DBMS Reliability:

A reliable distributed database management system is one that can continue to process user requests even when the underlying system is unreliable. In other words, even when components of the distributed computing environment fail, a reliable distributed DBMS should be able to continue executing user requests without violating database consistency.

Reliability Concepts and Measures:

Too often, the terms *reliability* and *availability* are used loosely in literature. Even among the researchers in the area of reliable computer systems, the definitions of these terms sometimes vary. In this section, we give precise definitions of a number of concepts that are fundamental to an understanding and study of reliable systems. Our definitions follow those of Anderson and Lee [1985] and Randell et al. [1978]. Nevertheless, we indicate where these definitions might differ from other usage of the terms.

System, State, and Failure

Reliability refers to a *system* that consists of a set of *components*. The system has a *state*, which changes as the system operates. The behavior of the system in providing response to all the possible external stimuli is laid out in an authoritative *specification* of its behavior. The specification indicates the valid behavior of each system state.

Any deviation of a system from the behavior described in the specification is con- sidered a *failure*. For example, in a distributed transaction manager the specification may state that only serializable schedules for the execution of concurrent transactions should be generated. If the transaction manager generates a non-serializable schedule, we say that it has failed.

Each failure obviously needs to be traced back to its cause. Failures in a system can be attributed to deficiencies either in the components that make it up, or in the design, that is, how these components are put together. Each state that a reliable system goes through is valid in the sense that the state fully meets its specification. However, in an unreliable system, it is possible that the system may get to an internal state that may not obey its specification. Further transitions from this state would eventually cause a system failure. Such internal states are called *erroneous states*; the part of the state that is incorrect is called an *error* in the system. Any error in the internal states of the components of a system or in the design of a system is called a *fault* in the system. Thus, a fault causes an error that results in a system failure (Figure 1.1).

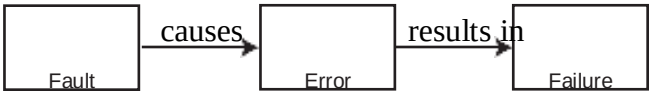


Fig. 1.1 Chain of Events Leading to System Failure

We differentiate between errors (or faults and failures) that are permanent and those that are not permanent. Permanence can apply to a failure, a fault, or an error, although we typically use the term with respect to faults. A *permanent fault*, also commonly called a *hard fault*, is one that reflects an irreversible change in the behavior of the system. Permanent faults cause permanent errors that result in permanent failures. The characteristic of these failures is that recovery from them requires intervention to "repair" the fault. Systems also experience *intermittent* and *transient faults*. In the literature, these two are typically not differentiated; they are jointly called *soft faults*. The dividing line in this differentiation is the repairability of the system that has experienced the fault [Siewiorek and Swarz, 1982]. An intermittent fault refers to a fault that demonstrates itself occasionally due to unstable hardware or varying hardware or software states. A typical example is the faults that systems may demonstrate when the load becomes too heavy. On the other hand, a transient fault describes a fault that results from temporary environmental conditions. A transient fault might occur, for example, due to a sudden increase in the room temperature. The transient fault is therefore the result of environmental conditions that may be impossible to repair. An intermittent fault, on the other hand, can be repaired since the fault can be traced to a component of the system.

These are the sources of a significant number of errors as the statistics included further in this section demonstrate. The relationship between various types of faults and failures is depicted in Figure 1.2.

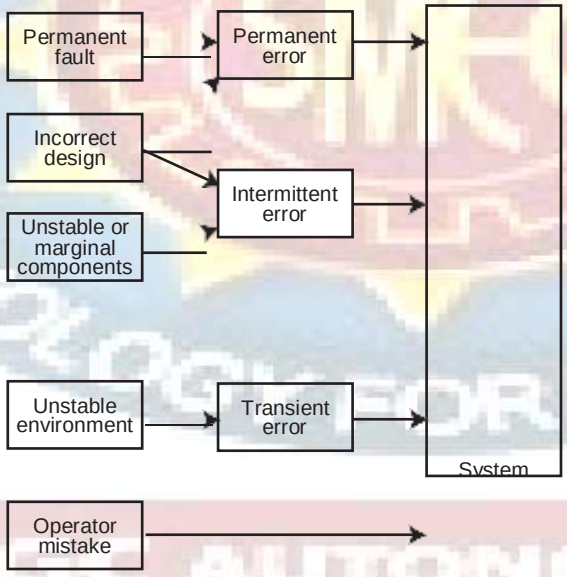


Fig. 1.2 Sources of System Failure

Reliability and Availability

Reliability refers to the probability that the system under consideration does not experience any failures in a given time interval. It is typically used to describe systems that cannot be repaired (as in space-based computers), or where the operation of the system is so critical that no downtime for repair can be tolerated.

Formally, the reliability of a system, $R(t)$, is defined as the following conditional probability:

$$R(t) = \Pr\{0 \text{ failures in time } [0, t] \mid \text{no failures at } t = 0\}$$

If we assume that failures follow a Poisson distribution (which is usually the case for hardware), this formula reduces to

$$R(t) = \Pr\{0 \text{ failures in time } [0, t]\}$$

Under the same assumptions, it is possible to derive that

$$\Pr\{k \text{ failures in time } [0, t]\} = \frac{e^{-m(t)}[m(t)]^k}{k!}$$

where $m(t) = \int_0^t z(x)\,dx$. Here $z(t)$ is known as the *hazard function*, which gives the time-dependent failure rate of the specific hardware component under consideration. The probability distribution for $z(t)$ may be different for different electronic components.

The expected (mean) number of failures in time $[0, t]$ can then be computed as

$$E[k] = \sum_{k=0}^{\infty} k \frac{e^{-m(t)}[m(t)]^k}{k!} = m(t)$$

and the variance as

$$Var[k] = E[k^2] - (E[k])^2 = m(t)$$

Given these values, $R(t)$ can be written as

$$R(t) = e^{-m(t)}$$

Note that the reliability equation above is written for one component of the system. For a system that consists of n non-redundant components (i.e., they all have to function properly for the system to work) whose failures are independent, the overall system reliability can be written as

$$R_{sys}(t) = \prod_{i=1}^{n} R_i(t)$$

Availability, $A(t)$, refers to the probability that the system is operational according to its specification at a given point in time t. A number of failures may have occurred

prior to time *t*, but if they have all been repaired, the system is available at time *t*. Obviously, availability refers to systems that can be repaired.

If one looks at the limit of availability as time goes to infinity, it refers to the expected percentage of time that the system under consideration is available to perform useful computations. Availability can be used as some measure of "goodness" for those systems that can be repaired and which can be out of service for short periods of time during repair. Reliability and availability of a system are considered to be contradictory objectives [Siewiorek and Swarz, 1982]. It is usually accepted that it is easier to develop highly available systems as opposed to highly reliable systems.

If we assume that failures follow a Poisson distribution with a failure rate λ , and that repair time is exponential with a mean repair time of $1/\mu$, the steady-state availability of a system can be written as

$$A = \frac{\mu}{\lambda + \mu}$$

Mean Time between Failures/Mean Time to Repair

Two single-parameter measures have become more popular than the reliability and availability functions given above to model the behavior of systems. These two measures used are *mean time between failures* (MTBF) and *mean time to repair* (MTTR). MTBF is the expected time between subsequent failures in a system with repair.[1] MTBF can be calculated either from empirical data or from the reliability function as

$$\text{MTBF} = \int_0^\infty R(t)\, dt$$

Since $R(t)$ is related to the system failure rate, there is a direct relationship between MTBF and the failure rate of a system. MTTR is the expected time to repair a failed system. It is related to the repair rate as MTBF is related to the failure rate. Using
these two metrics, the steady-state availability of a system with exponential failure and repair rates can be specified as

$$A = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

System failures may be *latent*, in that a failure is typically detected some time after its occurrence. This period is called *error latency*, and the average error latency time over a number of identical systems is called *mean time to detect* (MTTD).

Figure 1.3 depicts the relationship of various reliability measures with the actual occurrences of faults.

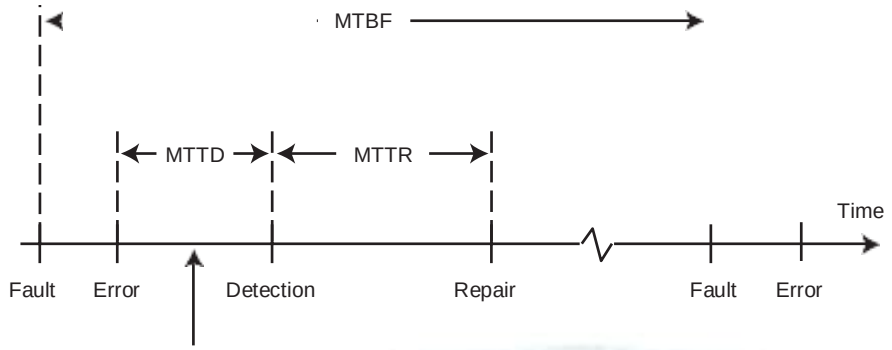


Fig. 1.3 Occurrence of Events over Time

Fault-tolerance in distributed systems:

Fault Tolerance is needed in order to provide 3 main feature to distributed systems. 1) Reliability- Focuses on a continuous service without any interruptions. 2) Availability - Concerned with read readiness of the system. 3) Security-Prevents any unauthorized access

What is a Fault?

Software fault is also known as defect, arises when the expected result don't match with the actual results. It can also be error, flaw, failure, or fault in a computer program. Most bugs arise from mistakes and errors made by developers, architects.

Fault Types

Following are the fault types associated with any:

- Business Logic Faults
- Functional and Logical Faults
- Faulty GUI
- Performance Faults
- Security Faults

Preventing Faults

Following are the methods for preventing programmers from introducing Faulty code during development:

- Programming Techniques adopted
- Software Development methodologies
- Peer Review
- Code Analysis

Failures in Distributed DBMS:

Designing a reliable system that can recover from failures requires identifying the types of failures with which the system has to deal. In a distributed database system, we need to deal with four types of failures: transaction failures (aborts), site (system) failures, media (disk) failures, and communication line failures. Some of these are due to hardware and others are due to software. The ratio of hardware failures vary from study to study and range from 18% to over 50%. Soft failures make up more than 90% of all hardware system failures. It is interesting to note that this percentage has not changed significantly since the early days of computing.

Transaction Failures

Transactions can fail for a number of reasons. Failure can be due to an error in the transaction caused by incorrect input data as well as the detection of a present or potential deadlock. Furthermore, some concurrency control algorithms do not permit a transaction to proceed or even to wait if the data that they attempt to access are currently being accessed by another transaction. This might also be considered a failure. The usual approach to take in cases of transaction failure is to *abort* the transaction, thus resetting the database to its state prior to the start of this transaction.[2]

Site (System) Failures

The reasons for system failure can be traced back to a hardware or to a software failure. The important point from the perspective of this discussion is that a system failure is always assumed to result in the loss of main memory contents. Therefore, any part of the database that was in main memory buffers is lost as a result of a system failure. However, the database that is stored in secondary storage is assumed to be safe and correct. In distributed database terminology, system failures are typically referred to as *site failures*, since they result in the failed site being unreachable from other sites in the distributed system.

We typically differentiate between partial and total failures in a distributed system. *Total failure* refers to the simultaneous failure of all sites in the distributed system; *partial failure* indicates the failure of only some sites while the others remain operational. As indicated in Chapter 1, it is this aspect of distributed systems that makes them more available.

Media Failures

Media failure refers to the failures of the secondary storage devices that store the database. Such failures may be due to operating system errors, as well as to hardware faults such as head crashes or controller failures. The important point from the perspective of DBMS reliability is that all or part of the database that is on the secondary storage is considered to be destroyed and inaccessible. Duplexing of disk storage and maintaining archival copies of the database are common techniques that deal with this sort of catastrophic problem.

Media failures are frequently treated as problems local to one site and therefore not specifically addressed in the reliability mechanisms of distributed DBMSs. We consider techniques for dealing with them in Section 12.3.5 under local recovery management. We then turn our attention to site failures when we consider distributed recovery functions.

Communication Failures

The three types of failures described above are common to both centralized and distributed DBMSs. Communication failures, however, are unique to the distributed case. There are a number of types of communication failures. The most common ones are the errors in the messages, improperly ordered

messages, lost (or undeliverable) messages, and communication line failures.

Network partitions point to a unique aspect of failures in distributed computer systems. In centralized systems the system state can be characterized as all-or- nothing: either the system is operational or it is not. Thus the failures are complete: when one occurs, the entire system becomes non-operational. Obviously, this is not true in distributed systems. As we indicated a number of times before, this is their potential strength.

Local & distributed reliability protocols:

In this section we discuss the functions performed by the local recovery manager (LRM) that exists at each site. These functions maintain the atomicity and durability properties of local transactions. They relate to the execution of the commands that are passed to the LRM, which are begin transaction, read, write, commit, and abort. Later in this section we introduce a new command into the LRM's repertoire that initiates recovery actions after a failure. Note that in this section we discuss the execution of these commands in a centralized environment. The complications introduced in distributed databases are addressed in the upcoming sections.

Architectural Considerations

It is again time to use our architectural model and discuss the specific interface between the LRM and the database buffer manager (BM). First note that the LRM is implemented within the data processor introduced in Chapter 11. The simple DP implementation that was given earlier will be enhanced with the reliability protocols discussed in this section. Also remember that all accesses to the database are via the database buffer manager. The detailed discussion of the algorithms that the buffer manager implements is beyond the scope of this book; we provide a summary later in this subsection. Even without these details, we can still specify the interface and its function, as depicted in Figure 1.4.[3]

In this discussion we assume that the database is stored permanently on secondary storage, which in this context is called the *stable storage* [Lampson and Sturgis, 1976]. The stability of this storage

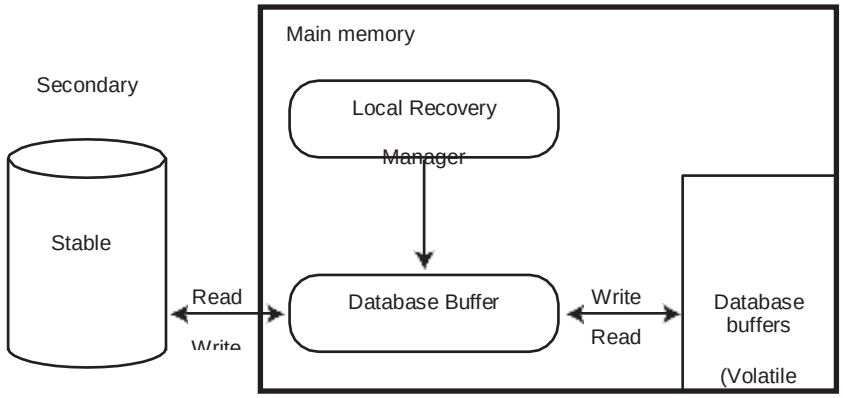


Fig. 1.4 Interface Between the Local Recovery Manager and the Buffer Manager

We should indicate that different LRM implementations may or may not use this forced writing. This issue is discussed further in subsequent sections.

As its interface suggests, the buffer manager acts as a conduit for all access to the database via the buffers that it manages. It provides this function by fulfilling three tasks:

1. *Searching* the buffer pool for a given page;
2. If it is not found in the buffer, *allocating* a free buffer page and *loading* the buffer page with a data page that is brought in from secondary storage;
3. If no free buffer pages are available, choosing a buffer page for *replacement*.

Searching is quite straightforward. Typically, the buffer pages are shared among the transactions that execute against the database, so search is global.

Allocation of buffer pages is typically done dynamically. This means that the allocation of buffer pages to processes is performed as processes execute. The buffer manager tries to calculate the number of buffer pages needed to run the process efficiently and attempts to allocate that number of pages. The best known dynamic allocation method is the *working-set algorithm* [Denning, 1968, 1980].

A second aspect of allocation is fetching data pages. The most common technique is *demand paging*, where data pages are brought into the buffer as they are referenced. However, a number of operating systems prefetch a group of data pages that are in close physical proximity to the data page referenced. Buffer managers choose this route if they detect sequential access to a file.

In replacing buffer pages, the best known technique is the least recently used (LRU) algorithm that attempts to determine the *logical reference strings* [Effelsberg and Härder, 1984] of processes to buffer pages and to replace the page that has not been referenced for an extended period. The anticipation here is that if a buffer page has not been referenced for a long time, it probably will not be referenced in the near future.

Recovery Information

In this section we assume that only system failures occur. We defer the discussion of techniques for recovering from media failures until later. Since we are dealing with centralized database recovery, communication failures are not applicable.

When a system failure occurs, the volatile database is lost. Therefore, the DBMS has to maintain some information about its state at the time of the failure in order to be able to bring the database to the state that it was in when the failure occurred. We call this information the *recovery information*.

The recovery information that the system maintains is dependent on the method of executing updates. Two possibilities are in-place updating and out-of-place updating. *In-place updating* physically changes the value of the data item in the stable database. As a result, the previous values are lost. *Out-of-place updating*, on the other hand, does not change the value of the data item in the stable database but maintains the new value separately. Of course, periodically, these updated values have to be integrated into the stable database. We should note that the reliability issues are somewhat simpler if in-place updating is not used. However, most DBMSs use it due to its improved performance.

In-Place Update Recovery Information

Since in-place updates cause previous values of the affected data items to be lost, it is necessary to keep enough information about the database state changes to facilitate the recovery of the database to a consistent state following a failure. This information is typically maintained in a *database log*. Thus each update transaction not only changes the database but the change is also recorded in the database log (Figure 1.5). The log contains information necessary to recover the database state following a failure.

Fig. 1.5 Update Operation Execution

Distributed Reliability Protocols:

As with local reliability protocols, the distributed versions aim to maintain the atomicity and durability of distributed transactions that execute over a number of
databases. The protocols address the distributed execution of the begin transaction, read, write, abort, commit, and recover commands.

At the outset we should indicate that the execution of the begin transaction, read, and write commands does not cause any significant problems. Begin transaction is executed in exactly the same manner as in the centralized case by the transaction manager at the originating site of the transaction. The read and write commands are executed as discussed in Chapter 11. At each site, the commands are executed in Similarly, abort is executed by undoing its effects.

Components of Distributed Reliability Protocols

The reliability techniques in distributed database systems consist of commit, termina- tion, and recovery protocols. Recall from the preceding section that the commit and recovery protocols specify how the commit and the recover commands are executed. Both of these commands need to be executed differently in a distributed DBMS than in a centralized DBMS. Termination protocols are unique to distributed systems. Assume that during the execution of a distributed transaction, one of the sites involved in the execution fails; we would like the other sites to terminate the transac- tion somehow. The techniques for dealing with this situation are called *termination protocols*. Termination and recovery protocols are two opposite faces of the recovery problem: given a site failure, termination protocols address how the operational sites deal with the failure, whereas recovery protocols deal with the procedure that the process (coordinator or participant) at the failed site has to go through to recover its state once the site is restarted. In the case of network partitioning, the termination protocols take the necessary measures to terminate the active transactions that exe- cute at different partitions, while the recovery protocols address the establishment of mutual consistency of replicated databases following reconnection of the partitions of the network.

The primary requirement of commit protocols is that they maintain the atomicity of distributed transactions. This means that even though the execution of the distributed transaction involves multiple sites, some of which might fail while executing, the effects of the transaction on the distributed database is all-or-nothing. This is called *atomic commitment*. We would prefer the termination protocols to be *non-blocking*. A protocol is non-blocking if it permits a transaction to terminate at the operational sites without waiting for recovery of the failed site. This would significantly improve the response-time performance of transactions. We would also like the distributed recovery protocols to be *independent*. Independent recovery protocols determine how to terminate a transaction that was executing at the time of a failure without having to consult any other site. Existence of such protocols would reduce the number of messages that need to be exchanged during recovery. Note that the existence of independent recovery protocols would imply the existence of non- blocking termination protocols, but the reverse is not true.

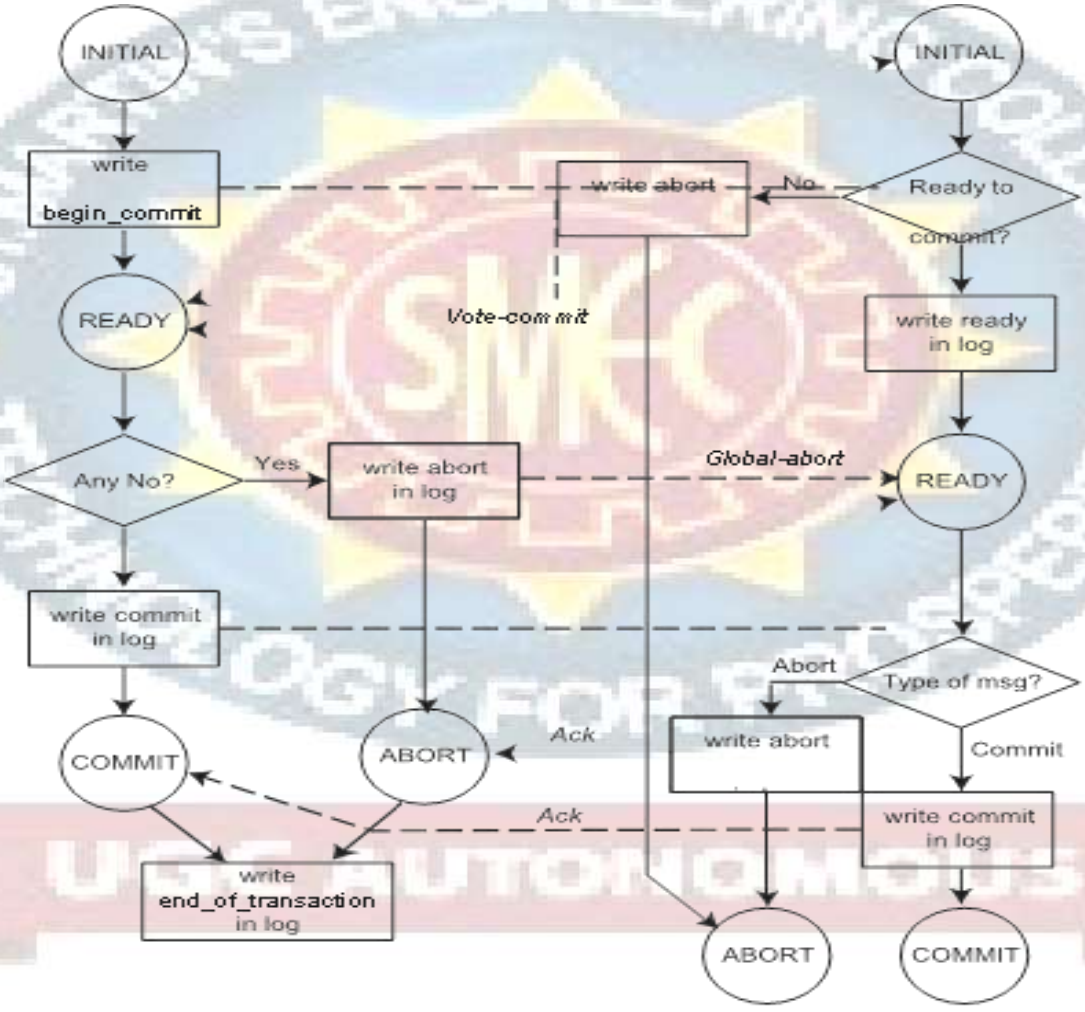
Two-Phase Commit Protocol

Two-phase commit (2PC) is a very simple and elegant protocol that ensures the atomic commitment of distributed transactions. It extends the effects of local atomic

commit actions to distributed transactions by insisting that all sites involved in the execution of a distributed transaction agree to commit the transaction before its effects are made permanent. There

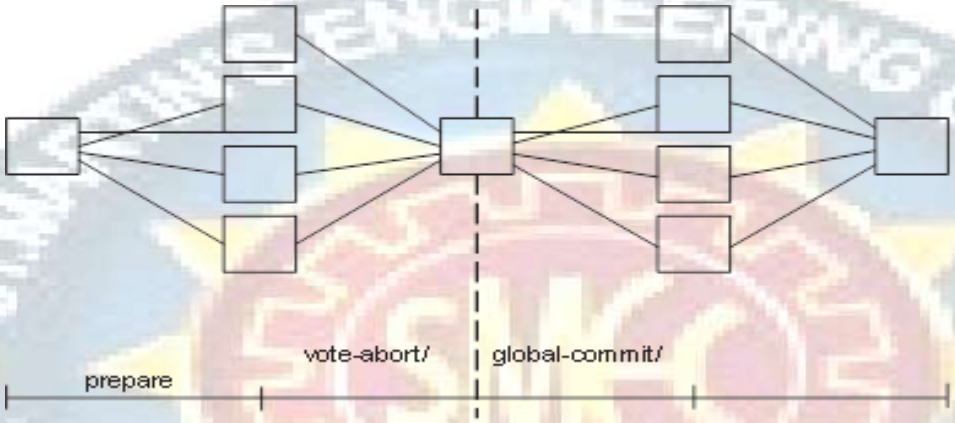
are a number of reasons why such synchronization among sites is necessary. First, depending on the type of concurrency control algorithm that is used, some schedulers may not be ready to terminate a transaction. For example, if a transaction has read a value of a data item that is updated by another transaction that has not yet committed, the associated scheduler may not want to commit the former. Of course, strict concurrency control algorithms that avoid cascading aborts would not permit the updated value of a data item to be read by any other transaction until the updating transaction terminates. This is sometimes called the *recoverability condition* ([Hadzilacos, 1988; Bernstein et al., 1987]).

Another possible reason why a participant may not agree to commit is due to deadlocks that require a participant to abort the transaction. Note that, in this case, the participant should be permitted to abort the transaction without being told to do so. This capability is quite important and is called *unilateral abort*.



There are a number of different communication paradigms that can be employed in implementing a 2PC protocol. The one discussed above and depicted in Figure 1.6 is called a *centralized 2PC* since the communication is only between the coordinator and the participants; the participants do not communicate among themselves. This communication structure, which is the basis of our subsequent discussions in this chapter, is depicted more clearly in Figure 1.7.

Coordinator Participants Coordinator Participants Coordinator



Centralized 2PC Communication Structure

Site failures and network partitioning:

In this section we consider the failure of sites in the network. Our aim is to develop non-blocking termination and independent recovery protocols. As we indicated before, the existence of independent recovery protocols would imply the existence of non-blocking recovery protocols. However, our discussion addresses both aspects

separately. Also note that in the following discussion we consider only the standard 2PC protocol, not its two variants presented above.

Termination and Recovery Protocols for 2PC:

Termination Protocols

The termination protocols serve the timeouts for both the coordinator and the par- ticipant processes. A timeout occurs at a destination site when it cannot get an expected message from a source site within the expected time period. In this section we consider that this is due to the failure of the source site.

The method for handling timeouts depends on the timing of failures as well as on the types of

failures. We therefore need to consider failures at various points of 2PC execution. This discussion is facilitated by means of the state transition diagram of the 2PC protocol.

Coordinator Timeouts.

There are three states in which the coordinator can timeout: WAIT, COMMIT, and ABORT. Timeouts during the last two are handled in the same manner. So we need to consider only two cases:

1. *Timeout in the WAIT state*. In the WAIT state, the coordinator is waiting for the local decisions of the participants. The coordinator cannot unilaterally commit the transaction since the global commit rule has not been satisfied.
   However, it can decide to globally abort the transaction, in which case it writes an abort record in the log and sends a "global-abort" message to all the participants.

(a) Coordinator (b) Participants

Fig. 1.7 State Transitions in 2PC Protocol

2. *Timeout in the COMMIT or ABORT states*. In this case the coordinator is not certain that the commit or abort procedures have been completed by the local recovery managers at all of the participant sites. Thus the coordinator
   repeatedly sends the "global-commit" or "global-abort" commands to the sites that have not yet responded, and waits for their acknowledgement.

Participant Timeouts.

A participant can time out[8] in two states: INITIAL and READY. Let us examine both of these cases.

1. *Timeout in the INITIAL state*. In this state the participant is waiting for a "prepare" message. The coordinator must have failed in the INITIAL state. The participant can unilaterally abort the transaction following a timeout. If
   the "prepare" message arrives at this participant at a later time, this can be handled in one of two possible ways. Either the participant would check its log, find the abort record, and respond with a "vote-abort," or it can simply ignore the "prepare" message. In the latter case the coordinator would time out in the WAIT state and follow the course we have discussed

above.

2. *Timeout in the READY state*. In this state the participant has voted to commit the transaction but does not know the global decision of the coordinator. The participant cannot unilaterally make a decision. Since it is in the READY state.

Coordinator Site Failures.

The following cases are possible:

1. *The coordinator fails while in the INITIAL state*. This is before the coordinator has initiated the commit procedure. Therefore, it will start the commit process upon recovery.
2. *The coordinator fails while in the WAIT state*. In this case, the coordinator has sent the "prepare" command. Upon recovery, the coordinator will restart the commit process for this transaction from the beginning by sending the "prepare" message one more time.
3. *The coordinator fails while in the COMMIT or ABORT states*. In this case, the coordinator will have informed the participants of its decision and terminated the transaction. Thus, upon recovery, it does not need to do anything if all the acknowledgments have been received. Otherwise, the termination protocol is involved.

Participant Site Failures.

There are three alternatives to consider:

1. *A participant fails in the INITIAL state*. Upon recovery, the participant should abort the transaction unilaterally. Let us see why this is acceptable. Note that the coordinator will be in the INITIAL or WAIT state with respect to this
transaction. If it is in the INITIAL state, it will send a "prepare" message and then move to the WAIT state. Because of the participant site's failure, it will not receive the participant's decision and will time out in that state. We have already discussed how the coordinator would handle timeouts in the WAIT state by globally aborting the transaction.
2. *A participant fails while in the READY state*. In this case the coordinator has been informed of the failed site's affirmative decision about the transaction before the failure. Upon recovery, the participant at the failed site can treat
this as a timeout in the READY state and hand the incomplete transaction over to its termination protocol.
3. *A participant fails while in the ABORT or COMMIT state*. These states represent the termination conditions, so, upon recovery, the participant does not need to take any special action.

Network Partitioning:

In this section we consider how the network partitions can be handled by the atomic commit protocols that we discussed in the preceding section. Network partitions are due to communication line failures and may cause the loss of messages, depending on the implementation of the communication subnet. A partitioning is called a *simple partitioning* if the network is divided into only two components; otherwise, it is called *multiple partitioning*.

The termination protocols for network partitioning address the termination of the transactions

that were active in each partition at the time of partitioning. If one can develop non-blocking protocols to terminate these transactions, it is possible for the sites in each partition to reach a termination decision (for a given transaction) which is consistent with the sites in the other partitions. This would imply that the sites in each partition can continue executing transactions despite the partitioning.

Unfortunately, it is not in general possible to find non-blocking termination protocols in the presence of network partitions. Remember that our expectations regarding the reliability of the communication subnet are minimal. If a message cannot be delivered, it is simply lost. In this case it can be proven that no non- blocking atomic commitment protocol exists that is resilient to network partitioning [Skeen and Stonebraker, 1983]. This is quite a negative result since it also means that if network partitioning occurs, we cannot continue normal operations in all partitions, which limits the availability of the entire distributed database system. A positive counter result, however, indicates that it is possible to design non-blocking atomic commit protocols that are resilient to simple partitions. Unfortunately, if multiple partitions occur, it is again not possible to design such protocols [Skeen and Stonebraker, 1983].

In the presence of network partitioning of non-replicated databases, the major concern is with the termination of transactions that were active at the time of par- titioning. Any new transaction that accesses a data item that is stored in another partition is simply blocked and has to await the repair of the network. Concurrent accesses to the data items within one partition can be handled by the concurrency control algorithm. The significant problem, therefore, is to ensure that the transaction terminates properly. In short, the network partitioning problem is handled by the commit protocol, and more specifically, by the termination and recovery protocols.

The absence of non-blocking protocols that would guarantee atomic commitment of distributed transactions points to an important design decision. We can either permit all the partitions to continue their normal operations and accept the fact that database consistency may be compromised, or we guarantee the consistency of the database by employing strategies that would permit operation in one of the partitions while the sites in the others remain blocked. This decision problem is the premise of a classification of partition handling strategies. We can classify the strategies as *pessimistic* or *optimistic* [Davidson et al., 1985]. Pessimistic strategies emphasize the consistency of the database, and would therefore not permit transactions to execute in a partition if there is no guarantee that the consistency of the database can be maintained. Optimistic approaches, on the other hand, emphasize the availability of the database even if this would cause inconsistencies.

The second dimension is related to the correctness criterion. If serializability is used as the fundamental correctness criterion, such strategies are called *syntactic* since the serializability theory uses only syntactic information. However, if we use a more abstract correctness criterion that is dependent on the semantics of the transactions or the database, the strategies are said to be *semantic*.

Consistent with the correctness criterion that we have adopted in this book (serial- izability), we consider only syntactic approaches in this section. The following two sections outline various syntactic strategies for non-replicated databases.

Parallel Database Systems: Parallel database system architectures

Parallel Database Architecture

Today everybody interested in storing the information they have got. Even small organizations collect data and maintain mega databases. Though the databases eat space, they really helpful in

many ways. For example, they are helpful in taking decisions through a decision support system. To handle such a voluminous data through conventional centralized system is bit complex. It means, even simple queries are time consuming queries. The solution is to handle those databases through Parallel Database Systems, where a table / database is distributed among multiple processors possibly equally to perform the queries in parallel. Such a system which share resources to handle massive data just to increase the performance of the whole system is called Parallel Database Systems.

We need certain architecture to handle the above said. That is, we need architectures which can handle data through data distribution, parallel query execution thereby produce good throughput of queries or Transactions. Figure 1, 2 and 3 shows the different architecture proposed and successfully implemented in the area of Parallel Database systems. In the figures, P represents Processors, M represents Memory, and D represents Disks/Disk setups.

1. Shared Memory Architecture

Figure 1 - Shared Memory Architecture

In Shared Memory architecture, single memory is shared among many processors as show in Figure 1. As shown in the figure, several processors are connected through an interconnection network with Main memory and disk setup. Here interconnection network is usually a high speed network (may be Bus, Mesh, or Hypercube) which makes data sharing (transporting) easy among the various components (Processor, Memory, and Disk).

Advantages:

Simple implementation

Establishes effective communication between processors through single memory addresses space. Above point leads to less communication overhead.

Disadvantages:

Higher degree of parallelism (more number of concurrent operations in different processors) cannot be achieved due to the reason that all the processors share the same interconnection network to connect with memory. This causes Bottleneck in interconnection network (Interference), especially in the case of Bus interconnection network.

Addition of processor would slow down the existing processors.

Cache-coherency should be maintained. That is, if any processor tries to read the data used or modified by other processors, then we need to ensure that the data is of latest version.

Degree of Parallelism is limited. More number of parallel processes might degrade the performance.

2. Shared Disk Architecture

Figure 2 - Shared Disk Architecture

In Shared Disk architecture, single disk or single disk setup is shared among all the available processors and also all the processors have their own private memories as shown in Figure 2.

Advantages:

Failure of any processors would not stop the entire system (Fault tolerance)
Interconnection to the memory is not a bottleneck. (It was bottleneck in Shared Memory architecture)
Support larger number of processors (when compared to Shared Memory architecture)
Disadvantages:

Interconnection to the disk is bottleneck as all processors share common disk setup.

Inter-processor communication is slow. The reason is, all the processors have their own memory. Hence, the communication between processors need reading of data from other processors' memory which needs additional software support.
Example Real Time Shared Disk Implementation
DEC clusters (VMScluster) running Rdb
3. Shared Nothing Architecture

Figure 3 - Shared Nothing Architecture

In Shared Nothing architecture, every processor has its own memory and disk setup. This setup may be considered as set of individual computers connected through high speed interconnection network using regular network protocols and switches for example to share data between computers. (This architecture is used in the Distributed Database System). In Shared Nothing parallel database system implementation, we insist the use of similar nodes that are Homogenous systems. (In distributed database System we may use Heterogeneous nodes)
Advantages:

Number of processors used here is scalable. That is, the design is flexible to add more number of computers.
Unlike in other two architectures, only the data request which cannot be answered by local processors need to be forwarded through interconnection network.
Disadvantages:

Non-local disk accesses are costly. That is, if one server receives the request. If the required data not available, it must be routed to the server where the data is available. It is slightly complex.
Communication cost involved in transporting data among computers.

Parallel data placement:
Parallel data placement refers to the physical placement of the data in a multiprocessor computer in order to favor parallel data access and yield high-performance. Most of the work on data placement has been done in the context of the shared-nothing architecture. Data placement in a parallel database system exhibits similarities with data fragmentation in distributed databases since fragmentation yields parallelism. However, there is no need to maximize local processing (at each node) since users are not associated with particular nodes and load balancing is much more difficult to achieve in the presence of a large number of nodes. The main solution for parallel data placement is a variation of horizontal fragmentation, called partitioning, which divides database relations into partitions, each stored at a different disk node. There are three basic partitioning strategies: round-

robin, hashing, and interval.

Parallel query processing:

The processing of a SQL statement is always performed by a single server process. With the parallel query feature, multiple processes can work together simultaneously to process a single SQL statement. This capability is called parallel query processing. By dividing the work necessary to process a statement among multiple server processes, the Oracle Server can process the statement more quickly than if only a single server process processed it.

The parallel query feature can dramatically improve performance for data-intensive operations associated with decision support applications or very large database environments. Symmetric multiprocessing (SMP), clustered, or massively parallel systems gain the largest performance benefits from the parallel query feature because query processing can be effectively split up among many CPUs on a single system.

It is important to note that the query is parallelized dynamically at execution time. Thus, if the distribution or location of the data changes, Oracle automatically adapts to optimize the parallelization for each execution of a SQL statement.

The parallel query feature helps systems scale in performance when adding hardware resources. If your system's CPUs and disk controllers are already heavily loaded, you need to alleviate the system's load before using the parallel query feature to improve performance. Chapter 18, "Parallel Query Tuning" describes how your system can achieve the best performance with the parallel query feature.

The Oracle Server can use parallel query processing for any of these statements:

- SELECT statements
- subqueries in UPDATE, INSERT, and DELETE statements
- CREATE TABLE ... AS SELECT statements
- CREATE INDEX

Parallel Query Process Architecture

Without the parallel query feature, a server process performs all necessary processing for the execution of a SQL statement. For example, to perform a full table scan (for example, SELECT * FROM EMP), one process performs the entire operation. The following figure illustrates a server process performing a full table scan:

Figure C-1: Full Table Scan without the Parallel Query feature

The parallel query feature allows certain operations (for example, full table scans or sorts) to be performed in parallel by multiple query server processes. One process, known as the query coordinator, dispatches the execution of a statement to several query servers and coordinates the results from all of the servers to send the results back to the user.

The following figure illustrates several query server processes simultaneously performing a partial scan of the EMP table. The results are then sent back to the query coordinator, which assembles the pieces into the desired full table scan.

Figure C-2: Multiple Query Servers Performing a Full Table Scan in Parallel

The query coordinator process is very similar to the server processes in previous releases of the Oracle Server. The difference is that the query coordinator can break down execution functions into parallel pieces and then integrate the partial results produced by the query servers. Query servers get assigned to each operation in a SQL statement (for example, a table scan or a sort operation), and the number of query servers assigned to a single operation is the degree of parallelism for a query.

The query coordinator calls upon the query servers during the execution of the SQL statement (not during the parsing of the statement). Therefore, when using the parallel query feature with the multi-threaded server, the server processing the EXECUTE call of a user's statement becomes the

query coordinator for the statement.

CREATE TABLE ... AS SELECT in Parallel

Decision support applications often require large amounts of data to be summarized or "rolled up" into smaller tables for use with ad hoc, decision support queries. Rollup often must occur regularly (such as nightly or weekly) during a short period of system inactivity. Because the summary table is derived from other tables' data, the recoverability from media failure for the smaller table may or may not be important and can be turned off. The parallel query feature allows you to parallelize the operation of creating a table as a subquery from another table or set of tables.

The following figure illustrates creating a table from a subquery in parallel.

Figure C-3: Creating a Summary Table in Parallel

If you disable recoverability during parallel table creation, you should take a backup of the tablespace containing the table once the table is created to avoid loss of the table due to media failure. For more information about recoverability of tables created in parallel, see the Oracle7 Server Administrator's Guide.

Clustered tables cannot be created and populated in parallel.

For a discussion of the syntax of the CREATE TABLE command, see the Oracle7 Server SQL Reference.

When creating a table in parallel, each of the query server processes uses the values in the STORAGE clause. Therefore, a table created with an INITIAL of 5M and a PARALLEL DEGREE of 12 consumes at least 60M of storage during table creation because each process starts with an extent of 5M. When the query coordinator process combines the extents, some of the extents may be trimmed, and the resulting table may be smaller than the requested 60M.

For more information on how extents are allocated when using the parallel query feature, see Oracle7 Server Concepts.

Parallelizing SQL Statements

When a statement is parsed, the optimizer determines the execution plan of a statement. Optimization is discussed in Chapter 7, "Optimization Modes and Hints". After the optimizer determines the execution plan of a statement, the query coordinator process determines the parallelization method of the statement. Parallelization is the process by which the query coordinator determines which operations can be performed in parallel and then enlists query server processes to execute the statement. This section tells you how the query coordinator process decides to parallelize a statement and how the user can specify how many query server processes can be assigned to each operation in an execution plan (that is, the degree of parallelism).

To decide how to parallelize a statement, the query coordinator process must decide whether to enlist query server processes and, if so, how many query server processes to enlist. When making these decisions, the query coordinator uses information specified in hints of a query, the table's definition, and initialization parameters. The precedence for selecting the degree of parallelism is described later in this section. It is important to note that the optimizer attempts to parallelize a query only if it contains at least one full table scan operation.

Each query undergoes an optimization and parallelization process when it is parsed. Therefore, when the data changes, if a more optimal execution plan or parallelization plan becomes available, Oracle can automatically adapt to the new situation.

In the case of creating a table in parallel, the subquery in the CREATE TABLE statement is parallelized and the actual population of the table is parallelized, as well as any enforcement of NOT NULL or CHECK constraints.

Parallelizing Operations

Before enlisting query server processes, the query coordinator process examines the operations in the execution plan to determine whether the individual operations can be parallelized. The Oracle Server can parallelize these operations:

- sorts
- joins
- table scans
- table population
- index creation

Partitioning Rows to Each Query Server

The query coordinator process also examines the partitioning requirements of each operation. An operation's partitioning requirement is the way in which the rows operated on by the operation must be divided, or partitioned, among the query server processes. The partitioning can be any of the following:

- range
- hash
- round robin
- random

After determining the partitioning requirement for each operation in the execution plan, the query coordinator determines the order in which the operations must be performed. With this information, the query coordinator determines the data flow of the statement. Figure C-4 illustrates the data flow of the following query:

```
SELECT dname, MAX(sal), AVG(sal)

        FROM emp, dept

                WHERE emp.deptno = dept.deptno

                GROUP BY dname;
```

Figure C-4: Data Flow Diagram for a Join of the EMP and DEPT Tables

Operations that require the output of other operations are known as parent operations. In Figure C-4 the GROUP BY SORT operation is the parent of the MERGE JOIN operation because GROUP BY SORT requires the MERGE JOIN output.

Parallelism Between Operations

Parent operations can begin processing rows as soon as the child operations have produced rows for the parent operation to consume. In the previous example, while the query servers are producing rows in the FULL SCAN DEPT operation, another set of query servers can begin to perform the MERGE JOIN operation to consume the rows. When the FULL SCAN DEPT operation is complete, the FULL SCAN EMP operation can begin to produce rows.

Figure C-5: Inter-Operator Parallelism and Dynamic Partitioning

Each of the two operations performed concurrently is given its own set of query server processes. Therefore, both query operations and the data flow tree itself have degrees of parallelism. The degree of parallelism of an individual operation is called intra-operation parallelism and the degree of parallelism between operations in a data flow tree is called inter-operation parallelism.

Due to the producer/consumer nature of the Oracle Server's query operations, only two operations in a given tree need to be performed simultaneously to minimize execution time.

To illustrate intra-operation parallelism and inter-operator parallelism, consider the following statement:

```
SELECT * FROM emp ORDER BY ename;
```

The execution plan consists of a full scan of the EMP table followed by a sorting of the retrieved rows based on the value of the ENAME column. For the sake of this example, assume the ENAME column is not indexed. Also assume that the degree of parallelism for the query is set to four, which

means that four query servers can be active for any given operation. Figure C-5 illustrates the parallel execution of our example query.

As you can see from Figure C-5, there are actually eight query servers involved in the query even though the degree of parallelism is four. This is because a parent and child operator can be performed at the same time. Also note that all of the query servers involved in the scan operation send rows to the appropriate query server performing the sort operation. If a row scanned by a query server contains a value for the ENAME column between A and G, that row gets sent to the first ORDER BY query server. When the scan operation is complete, the sorting query servers can return the sorted results to the query coordinator, which in turn returns the complete query results to the user.

Note: When a set of query servers completes its operation, it moves on to operations higher in the data flow. For example, in the previous diagram, if there was another ORDER BY operation after the ORDER BY, the query servers performing the table scan perform the second ORDER BY operation after completing the table scan.

Setting the Degree of Parallelism

The query coordinator process may enlist two or more of the instance's query server processes to process the statement. The number of query server processes associated with a single operation is known as the degree of parallelism. The degree of parallelism is specified at the query level (with hints), at the table level (in the table's definition), or by default in the initialization parameter file. Note that the degree of parallelism applies only to the intra-operation parallelism. If inter-operation parallelism is possible, the total number of query servers can be twice the specified degree of parallelism.

Determining the Degree of Parallelism for Operations

The query coordinator determines the degree of parallelism by considering three specifications. The query coordinator first checks for query hints, then looks at the table's definition, and finally checks initialization parameters for the instance for the default degree of parallelism. Once a degree of parallelism is found in one of these specifications, it becomes the degree of parallelism for the query.

For queries involving more than one table, the query coordinator requests the greatest number specified for any table in the query. For example, on a query joining the EMP and DEPT tables, if EMP's degree of parallelism is specified as 5 and DEPT's degree of parallelism is specified as 6, the query coordinator would request six query servers for each operation in the query.

Keep in mind that no more than two operations can be performed simultaneously. Therefore, the maximum number of query servers requested for any query can be up to twice the degree of parallelism per instance.

Hints, the table definitions, or initialization parameters only determine the number of query servers that the query coordinator requests for a given operation. The actual number of query servers used depends upon how many query servers are available in the query server pool and whether inter-operation parallelism is possible.

When you create a table and populate it with a subquery in parallel, the degree of parallelism for the population is determined by the table's degree of parallelism. If no degree of parallelism is specified in the newly created table, the degree of parallelism is derived from the subquery's parallelism. If the subquery cannot be parallelized, the table is created serially.

Hints

Hints allow you to set the degree of parallelism for a table in a query and the caching behavior of the query. Refer to Chapter 7, "Optimization Modes and Hints", for a general discussion on using hints in queries and the specific syntax for the PARALLEL, NOPARALLEL, CACHE, and NOCACHE hints.

Table and Cluster Definition Syntax

You can specify the degree of parallelism within a table definition. Use the CREATE TABLE,

ALTER TABLE, CREATE CLUSTER, or ALTER CLUSTER statements to set the degree of parallelism for a table or clustered table. Refer to the Oracle7 Server SQL Reference for the complete syntax of those commands.

Default Degree of Parallelism

Oracle determines the number of disks that the table is stored on and the number of CPUs in the system and then selects the smaller of these two values as the default degree of parallelism. The default degree of parallelism is used when you do not specify a degree of parallelism in a hint or within a table's definition.

For example, your system has 20 CPUs and you issue a parallel query on a table that is stored on 15 disk drives. The default degree of parallelism for your query is 15 query servers.

Note: The parameters PARALLEL_DEFAULT_SCANSIZE and PARALLEL_DEFAULT_MAX_SCANS are obsolete in release 7.3.

Minimum Number of Query Servers

Oracle can perform a query in parallel as long as there are at least two query servers available. If there are too few query servers available, your query may execute slower than expected. You can specify that a minimum percentage of requested query servers must be available in order for the query to execute. This ensures that your query executes with a minimum acceptable parallel query performance. If the minimum percentage of requested servers are not available, the query does not execute and returns an error.

Specify the desired minimum percentage of requested query servers with the initialization parameter PARALLEL_QUERY_MIN_PERCENT. For example, if you specify 50 for this parameter, then at least 50% of the query servers requested for any parallel operation must be available in order for the operation to succeed. If 20 query servers are requested, then at least 10 must be available or an error is returned to the user. If the value of PARALLEL_QUERY_MIN_PERCENT is set to null, then all parallel operations will proceed as long as at least two query servers are available for processing.

Limiting the Number of Available Instances

The INSTANCES keyword of the CREATE/ALTER TABLE/CLUSTER commands allows you to specify that a table or cluster is split up among the buffer caches of all available instances of an Oracle Parallel Server. If you do not want tables to be dynamically partitioned among all the available instances, you can specify the number of instances that can participate in scanning or caching with the parameter PARALLEL_DEFAULT_MAX_INSTANCES or the ALTER SYSTEM command.

If you want to specify the number of instances to participate in parallel query processing at startup time, you can specify a value for the initialization parameter PARALLEL_DEFAULT_MAX_INSTANCES. See the Oracle7 Server Reference for more information about this parameter.

If you want to limit the number of instances available for parallel query processing dynamically, use the ALTER SYSTEM command. For example, if you have ten instances running in your Parallel Server, but you want only eight to be involved in parallel query processing, you can specify a value by issuing the following command:

ALTER SYSTEM SET SCAN_INSTANCES = 8;

Therefore, if a table's definition has a value of ten specified in the INSTANCES keyword, the table will be scanned by query servers on eight of the ten instances. Oracle selects the first eight instances in this example. Set the parameter PARALLEL_MAX_SERVERS to zero on the instances that you do not want to participate in parallel query processing.

If you wish to limit the number of instances that cache a table, you can issue the following command:

ALTER SYSTEM SET CACHE_INSTANCES = 8;

Therefore, if a table specifies the CACHE keyword with the INSTANCES keyword specified as 10, it will divide evenly among eight of the ten available instances' buffer caches.

Managing the Query Servers

When you start your instance, the Oracle Server creates a pool of query server processes available for any query coordinator. The number of query server processes that the Oracle Server creates at instance startup is specified by the initialization parameter PARALLEL_MIN_SERVERS.

Query server processes remain associated with a statement throughout its execution phase. When the statement is completely processed, its query server processes become available to process other statements. The query coordinator process returns any resulting data to the user process issuing the statement.

Load balancing:

Distributed database applications needs to replicate data to improve data availability and query response time. Performance is improved because the fragment replica is stored at the nodes where they are frequently needed. Load balancing by proper allocation of transaction, and replicas by sharing of resources for performance analysis is important consideration during early stages. In this paper it is proposed a methodology for performance analysis of load balancing by sharing of resources, allocation of fragment replicas and transaction in distributed database system. The proposed methodology is simulated and results are validated using case study.

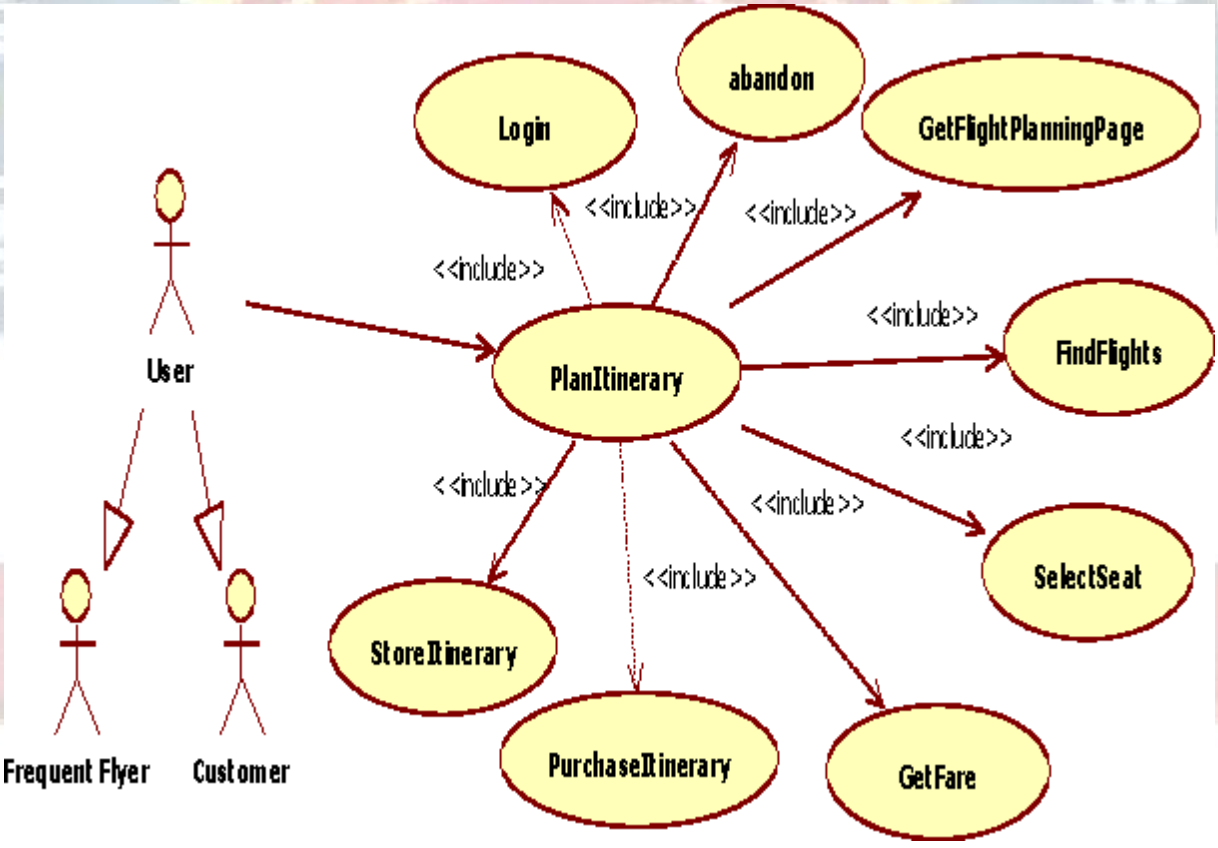


Fig: Load Balancing in Distributed Database System using Resource Allocation Approach

Database clusters:

A database cluster (DBC) is as a standard computer cluster (a cluster of PC nodes) running a Database Management System (DBMS) instance at each node. A DBC middleware is a software layer between a database application and the DBC. Such middleware is responsible for providing parallel query processing on top of the DBC.

It intercepts queries from applications and coordinates distributed and parallel query execution by taking advantage of the DBC. The DBC term comes from an analogy with the term PC cluster, which is a solution for parallel processing by assembling sequential PCs. In a PC cluster there is no need for special hardware to provide parallelism as opposed to parallel machines or supercomputers. A DBC takes advantage of off-the-shelf sequential DBMS to run parallel queries. There is no need for special software or hardware as opposed to parallel database systems.