Real Time Databases

Many real-time applications need to store large amounts of data and process these data for their successful operation. Such storing requirements occur when a controlling system needs to maintain an uptodate state of the controlled system. A few examples of such systems include a network management system, an industrial control system, and an autopilot system. Whenever storing and processing large amounts of data is required, a database management system is used. The need for a database management system for storage and processing of large volumes of data and the basic issues in relational database management systems (RDBMS) have been profusely discussed in standard database literature such as [1, 2, 3] and we assume that the reader is already familiar with these issues. The focus of this chapter is real-time database management systems (RTDBMS) that are used in data intensive real-time applications, such as network management systems, industrial control systems, autopilot systems, etc.

As traditional database systems do, real-time database systems also serve as repositories of large volumes of data and provide efficient storage, retrieval, and manipulation of data. However, there are a few important differences between traditional and real-time databases.

> The main differences between a conventional database and a real-time database lie in the temporal characteristics of the stored data, timing constraints imposed on the database operations, and the performance goals.

We elaborate these issues in this chapter. It would become clear that these issues make design and development of a satisfactory real-time database application much more difficult and complicated compared to traditional database application.

This chapter has been organized as follows. We first briefly examine a few applications needing support of a real-time database. Next, we review some of the basic concepts in traditional database technique that are relevant to the discussions in this Chapter. Subsequently, we elaborate implications of the temporal characteristics of data in a real-time database. Finally, we discuss some concurrency control protocols that can be used in real-time databases.

1 Example Applications of Real-Time Databases

In this section we review a few sample applications that need to use real-time databases. These applications are the representatives of a cross-section of applications having stringent timing requirements. An understanding of these applications would let us view the issues addressed in the subsequent sections in a proper perspective.

Process Control. As already discussed in Example 1 of Sec 1.1, industrial control systems are usually attached to sensors and actuators. The sensors monitor the state of some real-world processes, and the controllers manipulate the valves. Control decisions are made based on the input data and controller configuration parameters. Input data are generated from field devices such as sensors, transmitters, and switches that are connected to the controller's data acquisition interfaces, and also from other controllers via inter-controller connections. The input data and the controller configuration parameters run into several megabytes of information even for moderately large systems. For successful operation of a typical process control application, the valve control has to be achieved with the accuracy of a few milli seconds. Therefore, the database transactions have to be completed within a few milliseconds, even

under worst-case load on the system.

Internet Service Management. Internet traffic has grown phenomenally in the last few years. Also, advanced network services are increasingly becoming available as the Internet service providers (ISPs) vie to maintain their edge over each other. Toward this end, service providers are increasingly deploying service management systems (SMS). An SMS lets an ISP create and provide IP services such as e-mail, VPN, LDAP directory services, etc. The SMS streamlines allocation of resources to subscribers, resource management, and controlling all the relevant network components at a single centralized point in the network. An SMS needs to use a real-time database for performing authorization, authentication, and accounting for the Internet users. The SMS must manage such data as session status as well as information about the network, subscriber, and policies — in the face of the number of subscribers running into millions. Real-time database management systems are used to maintain these data.

Spacecraft Control System. In a spacecraft, a control system is responsible for the successful overall operation of the spacecraft. It is also responsible for receiving command and control information from the ground computer. A spacecraft control system maintains contact with the ground control using antennae, receivers, and transmitters. The control system monitors several parameters relevant to the successful operation of the spacecraft through several sensors mounted on and within the spacecraft. In addition to controlling the regular operation of the spacecraft, the controller also monitors the "health" of the spacecraft itself (power, telemetry, etc.). The reliability requirements for such systems dictate redundancy in all hardware and software components — increasing the volume of data and adding to the complexity of data management. The controlling information maintained by the controller include the track information. The volume of data maintained by this kind of applications is relatively small and restricted to a few megabytes of information, but the timing and performance attributes are very stringent.

Network Management System. A network management system for modern networks can be quite complex. The network management system stores and deals with large amounts of data pertaining to on network topology, configuration, equipment settings, etc. In addition, switches create large amounts of data pertaining to network traffic and faults. The network control, management, and administration operations lead to several real-time transactions on the database for storing and accessing the relevant data.

The database requirements for the above example applications vary in their timing requirements, from microseconds to make routing decisions in a network management system, to milliseconds for opening and closing of valves in an industrial process control application, and seconds for materials movement on a factory floor. However, irrespective of the magnitude of the timing constraint, what is more important is that unless the transaction timing constraints are met, the system would fail.

2 Review of Basic Database Concepts

In this section we review a few relevant database concepts. However, as already remarked these topics are widely covered in standard database literature such as [1, 2, 3], and the review in this section is by no means intended to be comprehensive. A relational database consists of a set of fact tables. Each fact table consists of several records. In many applications, it is required that the database records are prevented from assuming certain values or certain combination of values. Assertions about the values that the records can assume are called *consistency constraints.*

Database applications are normally structured into transactions. A transaction is a sequence of reads and writes on the database to achieve some high-level function of the application. A database transaction transforms a database from one consistent state to another. Normally, different transactions on a database operate in an interleaved manner. Interleaving the access of different transactions to the database can remarkably improve the throughput and resource utilization of the database. Therefore, one important aim in the design of database systems is to maximize the number of transactions that can be active at a time. A particular sequencing of actions of different transactions is called a *database schedule.* However, concurrent execution of transactions can lead to some database schedules

that violate integrity of the database. Due to the possibility of violation of integrity, databases normally restrict concurrent execution of transactions through the use of *concurrency control protocols.*

Concurrency control protocols maintain the integrity of the data by requiring the transactions to satisfy four important properties known as ACID properties. The ACID properties are briefly explained in the following:

Atomicity: Either all or none of the operations of a transaction are performed. That is, all the operations of a transaction are together treated as a single indivisible unit.

Consistency: A transaction needs to maintain the integrity constraints on the database.

Isolation: Transactions are executed concurrently as long as they do not interfere in each other's computations.

Durability: All changes made by a committed transaction become permanent in the database, surviving any subsequent failures.

Let us now examine how ACID properties are ensured in a database in the presence of interleaved execution of transactions. While each transaction preserves consistency of the database at its boundaries, a transaction that fails to complete might cause the integrity of the database to be violated. Rollback protocols are used to ensure atomicity and durability properties when a transaction fails to complete. Isolation can be ensured through the use of locking and rollback protocols.

Let us examine how a rollback protocol works. If a transaction t_j reads a value that was written by an aborted transaction t_i, then t_j must be aborted to enforce the atomicity property. In other words, a transaction might need to be rolled back and restarted because it had read a value that was produced by some other transaction which got aborted. However, it must be remembered that roll backs can lead to cascaded aborts. In this context, rollbacks have important implications for real-time applications — rollbacks often imply undoing significant amounts of accomplished work — the resultant delay in redoing the work could make a transaction miss its deadline.

To ensure the durability property, once a transaction commits, it can not be aborted, neither its effects changed due to cascading aborts. Cascadeless aborts can be achieved by ensuring that every transaction reads only data values written by committed transactions.

3 Real-Time Databases

Before we understand the various issues associated with real-time databases, we must understand how a real-time database differs from a traditional database. There are three main counts on which these two types of databases differ. First, unlike traditional databases, timing constraints are associated with the different operations carried out on real-time databases. Second, real-time databases have to deal with temporal data compared to static data as in the case of traditional databases. Third, the performance metrics that are meaningful to the transactions of these two types of databases are very different. We now elaborate these three issues.

Temporal Data: Data whose validity is lost after the elapse of some prespecified time interval are called temporal data or perishable data. Examples of such data include the following.

- Consider the periodic data generated by a temperature sensor. The temperature sensor transmits sampled temperature data periodically to the controller, say every 100 milliseconds. As new temperature readings become available, the old temperature data become stale and are reduced to archival data.
- Consider stock market price quotations. As new price quotations come in, data pertaining to previous quotations becomes obsolete.

- Consider the controller of a fly-by-wire aircraft. The fly-by-wire aircraft is expected to travel along a predetermined path. Every few milliseconds, the controller receives the current altitude, velocity, and acceleration data from various sensors mounted on the aircraft. From the received data and the last computed position of the aircraft, it computes the current position of the aircraft and the deviation of the aircraft from the predetermined path. Here, the current position, altitude, velocity, and acceleration values are temporal data; whereas the data representing the predetermined path is a traditional non-temporal data. This example application shows that a real-time database would have to deal with both temporal as well as archival data. In such a system, a database operation might compute results by combining the values of several temporal as well as archival data items.

Timing Constraints on Database Operations: Tasks and transactions are similar abstractions in the sense that both are units of work as well as scheduling. However, unlike real-time tasks, a transaction execution might require many data records in exclusive mode. Also, while the execution times of tasks can reasonably be assumed to be deterministic, the transaction execution times are much more unpredictable, especially if disk accesses are required.

Performance Metric: The most common performance metric for all databases — real-time or not — is transaction response time. For traditional database systems, this characteristic boils down to the number of transactions completed per unit time. This measurement therefore is used heavily in optimizing the average response time for traditional (non-real-time) applications. For real-time databases on the other hand, the typical metric of interest is the number of transactions missing their deadlines per unit time.

3.1 Real-Time Database Application Design Issues

Design of a real-time database application is much more intricate than design of databases for non-real-time applications. Let us investigate the reasons behind this. Irrespective of whether real-time or not, database transactions have extensive data requirements. Therefore, in case of real-time databases, if the data is stored in a secondary storage, the delay in accessing the data can make a transaction miss its deadline. Also, it becomes almost impossible to predict the response time for transactions due to the intricate protocols such as concurrency control protocols, commit protocols, and recovery protocols used to maintain the consistency of the database. Secondly, roll backs can have cascading effects and can introduce unpredictable amounts of delay.

In the face of the above mentioned problems, it might appear that use of databases is impractical in hard real-time applications. However, several silver lines exist. First, use of an in-memory database can make many of the problems vanish. Further, it must be remembered that in real-time applications, the set of transactions are simple and are known before hand (e.g. periodic sensor update). These transactions are fixed in the sense that they use the same amount and types of data each time. Therefore, plans for effective resource usage can be made to achieve deterministic transaction executions.

4 Characteristics of Temporal Data

A typical real-time system consists of a controlled system (environment) and a controlling system (computer). In other words, the controlling system maintains an image of the environment through periodic polling of sensor data. On the other hand, the environment is dynamic in nature and keeps changing its state unpredictably. The data representing the current state of the environment is an example of temporal data and is highly perishable. As an example, consider an antimissile system. In this, the controller maintains accurate information about the state of the missile given by its position, velocity, and acceleration data. The trajectory of an incoming missile is unpredictable and the antimissile system must maintain the state of the missile given by its current position, velocity, and acceleration values at any time. The current information maintained by the controller should be consistent with the actual state of the environment. This leads to the notion of temporal consistency. The need to maintain consistency between actual data of the environment, and that perceived by the controlling system necessitates the notion of

temporal consistency. In subsection 7.4.1, we elaborate the notion of temporal consistency.

Since the values of a set of parameters of the environment are recorded in a database table again and again, temporal attributes of these data must be stored. In subsection 7.4.1, we elaborate how temporal data can be represented in a database. In addition to temporal data, a real-time database may also contain archival data such as the desirable environment state (e.g. robot path). Therefore, real-time databases should be able to handle both temporal as well as archival data. Many transactions might have to combine several temporal data and possibly some archival data to derive new data. For example, in a rocket, the sampled velocity, acceleration, and position values can be used with the pre-stored desired path data to track the path error.

4.1 Temporal Consistency

Temporal consistency of data requires the actual state of the environment and the state represented by the database be very close and in any case within the limits required by the application. Temporal consistency of data has the following two main requirements:

Absolute Validity: This is the notion of consistency between the environment and its reflection in the database given by the data collected by the system about the environment.

Relative Consistency: This is the notion of consistency among the data that are used to derive new data.

Before we examine these notions in more detail, let us examine how data items can be represented in a real-time database and the notion of a relative consistency set.

How to Represent Data Items in a Real-Time Database?
A data item d can be represented as a triplet d:(value,avi,timestamp). The three components of a data item d are denoted as $d_{value}, d_{avi}, \ and \ d_{timestamp}$; where $d_{timestamp}$ denotes the time when measurement of d took place; d_{avi} is the absolute validity interval for the data item d and represents time interval following the $d_{timestamp}$ during which the data item d is considered to have absolute validity; d_{value} represents the value recorded for d. For example, a data item d=(120, 5msec,100msec) represents the value of the data item to be 120, recorded at 100msec, with an absolute validity interval of 5msec.

Relative Consistency Set.
Consider a situation where a set of data items used to derive a new data. For the derived data items to be correct, the set of data items on which it is based must be relatively consistent with each other. For example, in an antimissile system, the current velocity and position of a missile can be used to predict its new position. In this case, it would be incorrect to use an earlier sampled position with the velocity value to determine the new position of the missile. In other words, relative consistency ensures that only contemporary data items are used to derive new data. The set of data items that are relatively consistent with each other, form a relative consistency set R. Each R is associated with a relative validity interval (rvi), denoted by R_{rvi}. The relative consistency of the data items in the relative consistency set can be determined by using R_{rvi} as explained below.

Based on the above discussions, we can now define the conditions for absolute and relative validity as follows:

Condition for Absolute Validity: A data item d is absolutely valid, iff $(Current \ \ time - d_{timestamp}) \leq d_{avi}$

Condition for Relative Consistency: A set R of data items is relatively consistent, iff $\forall d, \forall d' \in R \ \ |d_{timestamp} - d'_{timestamp}| \leq R_{rvi}$

Example 7.1: Given a temporal data item $d = (10, 2500msec, 100msec)$ and the value of current time as $2700msec$. Is the given data item absolutely valid?

Solution: It has been given that $d_{avi} = 100$. So, d is valid during the interval between 2500 and 2600. Hence, the given data item d is not absolutely valid at the time instant 2700 msec.

□

Example 7.2: Let a relative consistency set R be {temperature, pressure} and lat R_{rvi} be 2. (a) Are temperature={347°C,5 msec ,95 msec } and pressure={50 bar,10 msec,97 msec} relatively consistent?
(b) Are temperature={347°C,5 msec ,95 msec } and pressure={50bar,10msec,92msec} relatively consistent?

Solution: (a) temperature={347°C,5 msec ,95 msec } and pressure={50 bar,10 msec,97 msec} are relatively consistent.
(b) temperature={347°C,5 msec ,95 msec } and pressure={50bar,10msec,92msec} are not relatively consistent. □

Example 7.3: Given that a relative consistency set R={position,velocity, acceleration} and $R_{rvi} = 100msec$ and following data items: $Position = (25m, 2500msec, 200msec), Velocity = (300m/s, 2550msec, 300msec), Acceleration = (20m/s^2, 2425msec, 200msec)$, $Current\ \ time = 2600msec$. Are the given data items absolutely valid? Also, are they relatively consistent?

Solution: Position is absolutely valid as $(2600 - 2500) < 200$
Velocity is also absolutely valid as $(2600 - 2550) < 300$
Acceleration is also absolutely valid as $(2600 - 2425) < 200$
For relative consistency, we have to check whether the different data items are pair-wise consistent. It can be easily checked that the given set of data is not relatively consistent, since for velocity and acceleration: $(2550 - 2425) \not< 100$.

□

5 Concurrency Control in Real-Time Databases

Each database transaction usually involves accesses to several data items, using which it carries out the necessary processing. Each access to data items takes considerable time, especially if disk accesses are involved. This contributes to making Transactions to be of longer duration than a typical task execution in a non-database application. For improved throughput, it is a good idea to start the execution of a transaction as soon as the transaction becomes ready (that is, concurrently along with other transactions already under execution), rather than executing them one after the other. Concurrent transactions at any time are those which are active (i.e. started but not yet complete). The concurrent transactions can operate either in an interleaved or in a "truly concurrent" manner — it does not really matter. What is important for a set of transactions to be concurrent is that they are active at the same time. It is very unlikely to find a commercial database that does not execute its transactions concurrently. However, unless the concurrent transactions are properly controlled, they may produce incorrect results by violating some ACID properties, e.g. result recorded by one transaction is immediately overwritten by another. ACID properties were discussed in Sec. 7.2. The main idea behind *concurrency control* is to ensure non-interference (isolation and atomicity) among different transactions.

Concurrency control schemes normally ensure non-interference among transactions by restricting concurrent transactions to be *serializable*. A concurrent execution of a set of transactions is said to be serializable, iff the database operations carried out by them is equivalent to some serial execution of these transactions. In other words, concurrency control protocols allow several transactions to access a database concurrently, but leave the database consistent by enforcing serializability.

Concurrency control protocols usually adopt any of the following two types of approaches. Concurrency control can be achieved either by disallowing certain types of transactions from progressing, or by allowing all transactions

to progress without any restrictions imposed on them, and then pruning some of the transactions. These two types of concurrency control protocols correspond to pessimistic and optimistic protocols. In a pessimistic protocol, permission must be obtained by a transaction, before it performs any operation on a database object. Permissions to transactions to access data items is restricted normally through the use of some locking scheme. Optimistic schemes neglect such permission controls and allow the transactions to freely access any data item they require. However, at transaction commitment, a validation test is conducted to verify that all database accesses maintain serializability. In the following, we first discuss a traditional locking-based concurrency control protocol called 2PL that is being popularly used in commercial non-real-time database management systems. We then discuss how this protocol has been extended for real-time applications. Subsequently, we examine a few optimistic concurrency control protocols designed for real-time applications.

5.1 Locking-based Concurrency Control

We first discuss 2PL, a popular pessimistic concurrency control protocol and then examine how this protocol has been extended for real-time applications.

2PL: It is a pessimistic protocol that restricts the degree of concurrency in a database. In this scheme, the execution of a transaction consists of two phases: a growing phase and a shrinking phase. In the growing phase, locks are acquired by a transaction on the desired data items. Locks are released in the shrinking phase. Once a lock is released by a transaction, its shrinking phase starts, and no further locks can be acquired by the transaction.

A strict 2PL is the most common protocol implemented in commercial databases. This protocol imposes an additional restriction on 2PL in that a transaction can not release any lock until after it terminates (i.e. commits or aborts). That is, all acquired locks are returned by a transaction only after the transaction terminates or commits. Though, this simplifies implementation of the protocol, a strict 2PL is too "strict" and prevents concurrent executions which could have easily been allowed without causing any violation of consistency of the database. A strict 2PL therefore introduces extra delays.

The conventional 2PL is unsatisfactory for real-time applications for several reasons: possibility of priority inversions, long blocking delays, lack of consideration for timing information, and deadlock. A priority inversion can occur, when a low priority transaction is holding a lock on a data item, and a high priority transaction needing the data item waits until the low priority transaction releases the lock. A transaction might undergo long blocking delays, since any other other transaction which might have acquired the data before it would hold it till its completion, and most transactions are usually of long duration types.

In the following we illustrate how deadlocks can occur in 2PL using an example. Consider the following sequence of actions by two transactions T_1 and T_2 which need access to two data items d1 and d2.

```
T1:  Lock d1, Lock d2, Unlock d2, Unlock d1
T2:  Lock d2, Lock d1, Unlock d1, Unlock d2
```

Assume that T_1 has higher priority than T_2. T_2 starts running first and locks data item d2. After some time, T_1 locks d1 and then tries to lock d2 which is being held by T_2. As a consequence T_1 blocks, and T_2 needs to lock the data item d1 being held by T_1. Now, the transactions T_1 and T_2 are both deadlocked.

2PL-WP: 2PL-WP (wait promote) is a scheme proposed to over come some of the shortcomings of the pure 2PL protocol. This scheme can be written in pseudo code form as follows:

```
                    /* pri(T) denotes priority of transaction T*/
if(pri(TR)>Pri(TH)) then  /* TH holds the lock requested by TR */
          TR waits;
          TH inherits priority of TR;
else
```

```
        TR waits;
endif
```

It can be observed from the pseudo code that when a data item requested by a high priority transaction is being held by a lower priority transaction, then the low priority transaction inherits the priority of the high priority transaction. 2PL-WP therefore deploys a priority inheritance scheme. However, in 2PL-WP unlike a pure priority inheritance scheme, if a higher priority transaction is aborted while it is being blocked, the elevated priority transaction retains the elevated priority until its termination. This can lead to the following undesirable situation. Under high data contention situations, 2PL-WP would result in most of the transactions in the system executing at the same priority. In this situation, the behavior of a database deploying 2PL-WP would reduce to that of a conventional database using 2PL. However, under low load situations, 2PL-WP should perform better than 2PL.

2PL-HP: 2PL-HP (high priority) overcomes some of the problems with 2PL-WP. In 2PL-HP, when a transaction requests a lock on a data object held by a lower priority transaction in a conflicting mode, the lock holding lower priority transaction is aborted. This protocol is also known as priority abort (PA) protocol. This protocol can be expressed in the form of the following pseudo code:

```
if (no conflict) then TR accesses D /* TR is the requesting transaction */
       else                         /* transaction TH is holding the data item, */
                                    /* resolve the conflict as follows. */
if (Pri(TR)>Pri(TH)) then abort  TH
       else TR waits for the lock;  /* TR blocks */
```

2PL-HP in addition to being free from priority inversion, is also free from deadlocks.

Experimental results show that real-time concurrency control protocols based on 2PL-HP outperform protocols based on either 2PL or 2PL-WP. This result might appear to be unexpected, since under 2PL-HP work is wasted due to transaction abortions occurring whenever a higher priority task requires a resource locked by a lower priority transaction. However, the result can be justified from the following consideration. Unlike tasks sharing resources intermittently, transactions under 2PL once they acquire data, hold them until their completion or abortion. Thus, the data holding time of a transaction is comparable to the life time of the transaction which is usually quite long. Transactions with resource contention, therefore undergo serial execution under 2PL rather than serializable execution. This implies that chances of deadline misses by higher priority transactions under 2PL may be more than that in 2PL-HP. In 2PL though the chances of cascaded roll backs are less, but the average transaction execution times have been reported to be longer.

Priority Ceiling Protocol: Let us now discuss Priority Ceiling Protocol (PCP) for databases. Unlike PCP for resource sharing among tasks; PCP in database concurrency control does not make use of any priority inheritance. The main concept involved in this protocol is the establishment of a total priority ordering among all transactions. This protocol associates the following three values with every data object.

Read Ceiling: This value indicates the priority value of the highest priority transaction that may write to the data object.

Absolute Ceiling: This is the highest priority transaction that may read or write the data object.

Read-Write Ceiling: This value is defined dynamically at run time. When a transaction writes to a data object, the read-write ceiling is set equal to the absolute ceiling. However, read-write ceiling is set equal to the read ceiling for a read operation.

The priority ceiling rule is the following.

> A transaction requesting access to a data object is granted the same, if and only if the priority of the transaction requesting the data object is higher than the read-write ceiling of all data objects.

It can be shown that PCP is deadlock free and single blocking. Recollect that single blocking means that once a transaction starts executing after being blocked, it may not block again.

Let us analyze a few interesting properties of this protocol. A property of this protocol is that transactions with priorities that are lower than or equal to the read ceiling are not allowed to read the data objects even in a compatible mode. Such a pessimistic measure has been taken to ensure that a future high priority transaction will not block on low priority writers. Also, after a transaction writes a data item, no other transaction is permitted to either read or write to that data item until the original writer terminates.

Example 7.4: Assume that the read ceiling for a data item d is 20 and the absolute ceiling is 40. This means that highest priority among all the transactions that might read d is 20. After any transaction reads d, the read-write ceiling is set to 20. This prevents any transaction which needs to read d from accessing d. □

5.2 Optimistic Concurrency Control Protocols

Optimistic Concurrency Control (OCC) Protocol as the name suggests does not in any way prevent any transaction from accessing any data items it requires. However, a transaction is validated at the time of its commitment and any conflicting transactions are aborted.

The concept of *validation* (also known as *certification*) is central to all OCC protocols.

If there is little contention and interference among transactions, most transactions would be successfully validated. Under heavy load conditions however, there can be large number of transactions that fail the validation test and are aborted. This might lead to severe reduction in throughput and sharp increase in deadline misses. Thus, the performance of OCC protocols can show a marked drop at high loads. We now discuss a few important optimistic concurrency control protocols.

Forward OCC: In this protocol, transactions read and update data items freely, storing their updates into a private work place. These updates are made public only at transaction commit time. Before a transaction is allowed to commit, it has to pass a validation test. The validation test checks whether there is any conflict between the validating transaction and other transactions that have committed since the validating transaction started executing. A transaction is aborted, if it does not pass the validation test. This guarantees the atomicity and durability properties. Since writes occur only at commit time, the serialization order in OCC is the order in which the transactions commit.

OCC Broadcast Commit: In OCC Broadcast Commit (OCC-BC) protocol, when a transaction commits, it notifies its intention to all other currently running transactions. Each of these running transactions carries out a test to check if it has any conflicts with the committing transaction. If any conflicts are detected, then the transaction carrying out the check immediately aborts itself and restarts.

Note that there is no need for a committing transaction to check for conflicts with already committed transactions, because if it were in conflict with any of the committed transactions, it would have already been aborted. Thus, in OCC-BC once a transaction reaches its validating phase, it is guaranteed commitment. Compared to OCC-forward, it encounters earlier restarts and less wasted computations. Therefore, this protocol should perform better than the OCC-forward protocol in meeting task deadlines. However, a problem with this protocol is that it does not consider the priorities of transactions. On the other hand, it may be possible to achieve better performance by explicitly considering the priorities of the transactions.

OCC-Sacrifice: This protocol explicitly considers the priorities of transactions. Also, this protocol introduces the concept of a *conflict set*. A conflict set is the set of currently running transactions that conflict with the

validating transaction. A transaction once reaches its validation stage, checks for conflicts with the currently executing transactions. If conflicts are detected and one or more of the transactions in the conflict set has higher priority than the validating transaction, then the validating transaction is aborted and restarted. Otherwise, all transactions in the conflict set are aborted and restarted. However, this protocol suffers from the problem that a transaction may be sacrificed on behalf of another transaction that is sacrificed later, and so on — leading to wasted computations and deadline misses.

5.3 Speculative Concurrency Control

Speculative Concurrency Control (SCC) protocol tries to overcome a major weakness of the OCC protocols. Recollect that OCC protocols ignore the occurrence of conflict between two transactions until the validation phase of one of them. At this time, it may already have become too late to correct the problem by restarting one of the transactions, especially if the transactions have tight deadlines. To overcome this problem, in SCC, conflicts are checked at every read and write operations. Whenever a conflict is detected, a new version (called the shadow version) of each of the conflicting transactions is initiated. The primary version executes as any transaction would execute under an OCC protocol, ignoring the conflicts that develop. Meanwhile, the shadow version executes as any transaction would do under a pessimistic protocol — subjected to locking and restarts. The idea is to always keep a clean version (a version without any conflicts) in case it is ever needed. This should help in meeting the deadlines of tasks with tight deadlines. When a shadow reaches a point of conflict, it blocks as any transaction under a pessimistic protocol. When the primary version commits, any shadow associated with it is discarded. In addition, any shadow whose serializability depends on the discarded shadows is also discarded. However, when the primary transaction aborts, the shadow starts off executing under OCC as the new primary.

As in the OCC protocols, in SCC protocol also all updates made by a transaction are made on local copies and therefore are not visible until the transaction commits.

5.4 Comparison of Concurrency Control Protocols

Before we discuss the relative performance of the different categories of protocols, we highlight some aspects of concurrency control protocols that have significant bearings on their performance. This would help us understand the results obtained from experiments in proper perspective. Locking-based algorithms tend to reduce the degree of concurrent execution of transactions, as they construct serializable schedules. On the other hand, optimistic approaches attempt to increase parallelism to its maximum, but they prune some of the transactions in order to satisfy serializability. In 2PL-HP, a transaction could be restarted by, or wait for another transaction that will be aborted later. Such restarts and/or waits cause performance degradation.

Unlike 2PL-HP, in optimistic approaches when incorporating broadcast commit schemes (OCC-BC), only validating transactions can cause restart of other transactions. Since all validating transactions are guaranteed commitment at completion, all restarts generated by such optimistic protocols are useful [4]. Further, the OCC protocols have the advantage of being non-blocking and free from deadlocks — a very desirable feature in real-time applications.

Performance study results for the different categories of concurrency control protocols are shown in Fig. 7.1. As can be intuitively expected, at zero contention, all the three types of protocols result in identical performance. From Fig. 7.1 it can be seen that at low conflicts OCC outperforms pessimistic protocols. However, pessimistic protocols perform better as the load (and therefore the conflicts) become higher. On the other hand, SCC performs better compared to both OCC and pessimistic protocols, when transactions have tight deadlines and the load is moderate.

6 Commercial Real-Time Databases

A commercial real-time database needs to avoid using anything that can introduce unpredictable latencies. Thus, a commercial real-time database needs to avoid using disk I/O operations, message passing or garbage collection.

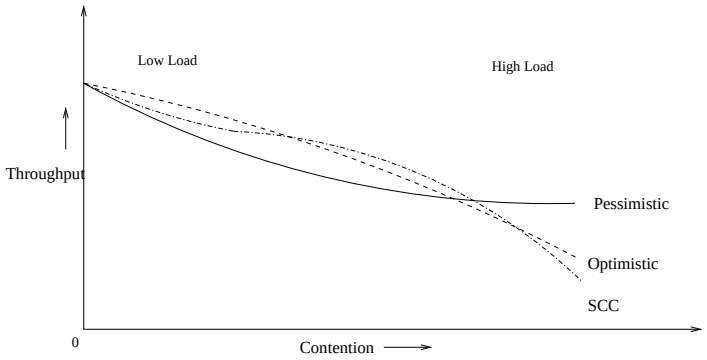

Figure 1: Performance of Different Categories of Concurrency Control Protocols

Real-time databases therefore tend to be designed as in-memory database systems. In-memory databases do away with disk I/O entirely, and their simplified design (compared to conventional databases) minimizes message passing. Similar to non-real-time disk-based databases, in-memory databases provide database integrity through the use of conventional disks for logging and periodic checkpoints. To meet the high availability demands of applications, some in-memory databases offer data replication.

An example of a commercial real-time database is McObject's eXtremeDB (http://www.mcobject.com). eXremeDb has successfully been used in applications such as telecommunication, factory automation, process control, consumer electronic devices, and medical equipments. eXtremeDB provides applications with direct access to data. That is, the database is mapped directly into the application's address space, eliminating expensive buffer managements. The eXtremeDB runtime component is directly linked to the application, eliminating the need for remote procedure calls from execution path. To provide performance and predictability, eXtremeDB uses its own memory manager for allocations and deallocations made by the database runtime. Not relying on the operating system's memory management also enables eXtremeDB to remove the bottleneck of paging data in and out during I/O operations. For supporting transaction deadline management, eXtremeDB supports five priority levels that can be assigned to transactions.

SUMMARY

- A transaction is a unit of execution in a database application, and is a concept similar to a task in a non-database application.
- Major differences between a task and a transaction include high data usage by transactions. Further, it is very difficult to predict transaction response times due to issues such as I/O scheduling (disk access), buffering, concurrency control, commit and recovery protocols used, etc. As a consequence, real-time databases used in hard real-time applications are rarely disk-based.
- Use of an appropriate concurrency protocol is important to meet transaction deadlines. The standard 2PL protocol when used in real-time databases results in many problems: long and unpredictable delays, unbounded priority inversions, and deadlocks.
- For real-time applications, an improvement of 2PL is 2PL-WP, but it behaves as badly when number of transactions and resource sharing are large. 2PL-HP overcomes some of the problems of 2PL-WP, but may result in wastage of large amounts of completed work.
- Optimistic concurrency control (OCC) protocols allow unrestricted data usage by transactions. However, they subject a committing transaction to a validation test at the time of commitment. These protocols work the best under low resource constraints and low load situations.

EXERCISES

1. State whether you consider the following statements to be **TRUE** or **FALSE**. Justify your answer in each case.
   (a) For temporal data used in a real-time application, a set of data that is absolutely consistent, is guaranteed to be relatively consistent.
   (b) The performance of the 2PL-WP protocol is better than the basic 2PL protocol when the data contention among transactions is low, but the performance becomes worse than the basic 2PL protocol under high data contention among transactions.
   (c) 2PL-WP protocol used in concurrency control in real-time databases is free from deadlocks.
   (d) 2PL-HP protocol used in concurrency control in real-time databases, is free from priority inversion and is also free from deadlocks.
   (e) Under the OCC-BC protocol used in concurrency control in real-time databases, once a transaction reaches the validation phase, it is guaranteed commitment.
   (f) In a real-time database management system, an optimistic concurrency control (OCC) protocol should be used if the choice is based purely on performance (percentage of tasks meeting their deadlines) considerations.
   (g) OCC-BC protocol requires a committing transaction to check for possible conflicts with already committed transactions before it can commit.
   (h) The serialization order under a pessimistic concurrency control protocol is the same as the committing order of the transactions.
   (i) Real-time databases typically use RAID disks for data storage during system operation.
   (j) Optimistic Concurrency Control (OCC) protocols used for concurrency control in real-time databases are free from deadlocks.
   (k) All OCC protocols essentially perform forward validations just before their commitment, i.e. a transaction validates itself with already completed transactions.
2. Explain how a real-time database differs from a conventional database.
3. Explain a few practical applications requiring the use of a real-time database.
4. What do you understand by *temporal data*? How are temporal data different from traditional data? Give some examples of temporal data.
5. Suppose, temporal data are denoted using triplets of the form {value, avi, timestamp} and that the different components of the temporal data have their usual meanings. Assume that the relative consistency set R={pressure,temperature} and $R_{rvi} = 2msec$. Suppose the temperature and pressure samples taken at some time instant are given by {223°C,5msec,112msec} and {77°C,10msec,114msec} respectively. At the time instant 120 msec, determine whether the temperature and pressure samples are i) absolutely consistent, ii) relatively consistent.
6. What is the role of a *concurrency control protocol* in a database? Why is selection of an appropriate concurrency control protocol important to meet the timeliness requirements for transactions? Explain the different categories of concurrency control protocols that can be used in real-time databases. Also, explain which category of protocol is best suited under what circumstances.
7. Why is a traditional 2 phase locking (2PL) based concurrency control protocol may not be suitable for use in real-time databases? Explain how the traditional 2PL protocol can be extended to make it suitable for use in real-time database applications.
8. For real-time applications, rank 2PL, SCC, and OCC protocols in terms of the percentage of transactions meeting their deadlines. Consider low and high degrees of conflicts among transactions, and tight and lax transaction deadlines. Give a brief reasoning behind your answer.

References

[1] Date C.J. *Database in Depth.* O'Reily, 2005.

[2] Ramakrishnan Raghu. *Database Management Systems.* McGraw-Hill, 2002.

[3] Abraham Silbershatz, Korth, and Sudarshan. *Database System Concepts.* McGraw-Hill, 2005.

[4] Song X. and Liu W.S. Maintaining temporal consistency: Pessimistic vs. optimistic concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):786–796, October 1995.

Source:http://www.nptel.ac.in/courses/Webcourse-contents/IIT%20Kharagpur/Real%20time%20system/pdf/Module2.pdf