AN OVERVIEW OF REAL-TIME DATABASE SYSTEMS

Ben Kao[1,2] *and Hector Garcia-Molina*[2]

[1] Princeton University, Princeton NJ 08544, USA
[2] Stanford University, Stanford CA 94305, USA

1 Introduction

Traditionally, real-time systems manage their data (e.g. chamber temperature, aircraft locations) in application dependent structures. As real-time systems evolve, their applications become more complex and require access to more data. It thus becomes necessary to manage the data in a systematic and organized fashion. Database management systems provide tools for such organization, so in recent years there has been interest in "merging" database and real-time technology. The resulting integrated system, which provides database operations with real-time constraints is generally called a real-time database system (RTDBS) [1].

Like a conventional database system, a RTDBS functions as a repository of data, provides efficient storage, and performs retrieval and manipulation of information. However, as a part of a real-time system, whose "tasks" are associated with time constraints, a RTDBS, has the added burden of ensuring some degree of confidence in meeting the system's timing requirements.

Example applications that handle large amounts of data and have stringent timing requirements include telephone switching (e.g. translating an 800 number into an actual number), radar tracking and others. Arbitrage trading, for example, involves trading commodities in different markets at different prices. Since price discrepancies are usually short-lived, automated searching and processing of large amounts of trading information are very desirable. In order to capitalize on the opportunities, buy-sell decisions have to be made promptly, often with a time constraint so that the financial overhead in performing the trade actions are well compensated by the benefit resulting from the trade. As another example, a radar surveillance system detects aircraft "images" or "radar signatures". These images are then matched against a database of known images. The result of such match is used to drive other system actions, for example, in choosing a combat strategy.

Conventional database systems are not adequate for this type of application. They differ from a RTDBS in many aspects. Most importantly RTDBSs have different performance goals, correctness criteria, and assumptions about the applications. Unlike a conventional database system, whose main objective is to provide fast "average" response time, a RTDBS may be evaluated based on how often transactions miss their deadlines, the average "lateness" or "tardiness" of late transactions, the cost incurred in transactions missing their deadlines, data external consistency (how current the values of data are in reflecting the state of the external world), and data temporal consistency (values of data in the database should be taken from the external world at similar times) [50].

As a real-time system, specifications related to timing constraints are usually supplied by the application designers. For most cases, these timing requirements are expressed as deadlines for transactions. Transactions of this sort, with which explicit time constraints are associated, are termed real-time transactions.

As mentioned above, a RTDBS can be viewed as a value-added database system that supports real-time transactions. A real-time transaction has to be completed by its deadline to be of full benefit to the system. Such guarantees are usually hard to ensure. In case a transaction's deadline is not met, the transaction is called a tardy transaction.

Real-time database systems differ in the way tardy transactions are handled, and this issue is generally referred to as the overload management problem. A tardy transaction may carry positive, zero, or negative residual value to the system. For the positive case, even though the benefit obtained by completing the tardy transaction is usually less than its full fledged value, the system should still complete it, if possible. The system may, however, choose to lower the transaction's priority so that non-tardy transactions are given preferential treatment, for example, in accessing system resources. When a tardy transaction completely loses its value (zero residual value case), it should be dropped to free system resources for the benefit of other transactions. Finally, when a tardy transaction carries negative value, the system may choose to raise the transaction's priority so that it can be completed as soon as possible to diminish the cost incurred due to it tardiness. On the other hand, the system may lower the transaction's priority or even drop it so that other transactions have a better chance of meeting their deadlines. The decision is dependent upon the application semantics. In the extreme case that a system cannot afford having a tardy transaction (e.g. in nuclear power plant control), the system is said to be a hard real-time database system; otherwise, if tardy transactions are tolerated even though they may be undesirable (e.g. arbitrage trading), we say that the system is a soft real-time database system.

It is argued in [51] that with current technology, it is very hard to provide an absolute guarantee on meeting transaction deadlines, and therefore, RTDBSs are mostly limited to soft real-time systems. There are several factors that make it hard for a RTDBS to meet all deadlines. Firstly, the executions of database transactions are usually data and resource dependent. To guarantee satisfaction of transaction deadlines requires enormous excess resource to accommodate the highest system load. Secondly, full transaction support involves many database protocols which are highly unpredictable in their execution times[3]. Concurrency control protocols, for example, often introduce blocking and restart of trans-

[3] For a brief account on real-time system predictability, see [52].

actions over resource contention. Thirdly, disk-based database systems interact heavily with the I/O subsystem. Problems such as disk seek time variation, buffer management and page faults, cause the average case and worst case execution times to differ widely. All these add to the unpredictability of transaction execution.

While difficulties for ensuring transactions meet their deadlines certainly exist, since most RTDBSs are used for highly specialized applications, special techniques may be applied to improve the system's real-time behavior. For example, if the database is small enough to fit into main memory, most of the I/O operations can be eliminated. This in turn, gets rid of the problem of page faults and I/O scheduling. We will discuss main memory database systems later in this chapter.

Also, in some real-time systems, "tasks" or transactions can be preanalyzed. Semantic properties of transactions and data may be known a priori. The knowledge of transaction runtime and resource requirements may lead to more effective scheduling and concurrency control protocols. As an example [37], in a conventional database system the number of constraints is assumed to be large. Checking them individually may be impractical, so instead serializability is used as the correctness criterion. However, "since real-time systems may have a fixed number of processes and the databases are statically structured, it may be feasible to specify a small set of integrity constraints which are most critical for the system's correctness. [37]" Specialized protocols may then be designed that allow non-serializable but consistent schedules [33].

In the rest of this chapter, we will discuss some problems concerning the design of a RTDBS. We will present some solutions as proposed by the research community. We will also examine the various components of a database system and discuss what features should be added to support real-time transactions.

2 Transaction Model

In this section, we look at the attributes of real-time transactions and discuss how they affect transaction design. In particular, we will discuss the issue of deadline assignment, and how semantic information can be used to help meeting the system's timing constraints.

The following types of information about transactions may be available and may be of use in scheduling and concurrency control:

1. Timing Constraints — E.g. deadlines.
2. Criticalness — It measures how critical it is that a transaction meets its deadline. Different transactions may have different criticalness. Note that criticalness is a different concept from deadline. A transaction may have a very tight deadline but missing it may not cause great harm to the system.
3. Value function — Related to a transaction's criticalness is its value function. A value function of a transaction measures how valuable it is to complete the transaction at some point in time after the transaction arrives. Some typical value functions are shown in Fig. 1.

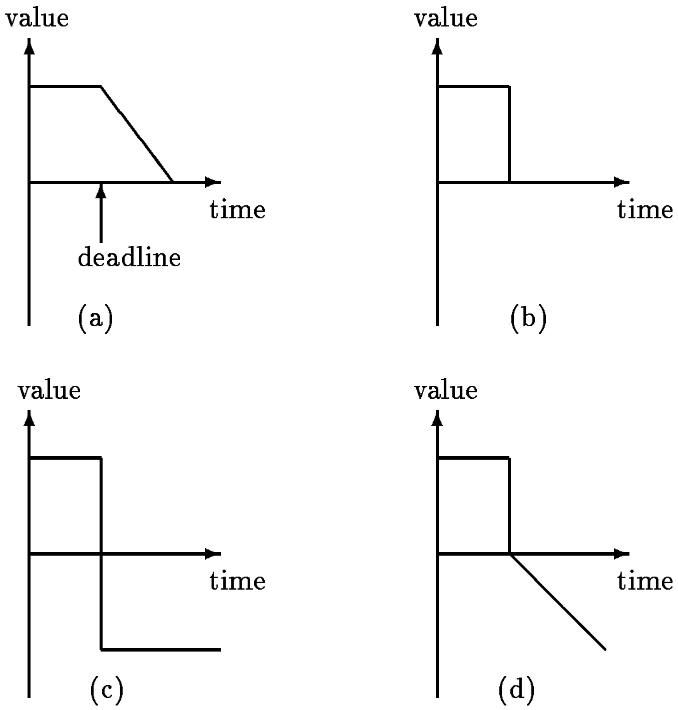

Fig. 1. Example value functions. Tardy transaction has (a) diminishing positive value, (b) zero value, (c) negative value, (c) increasingly negative value.

4. Resource requirements — This includes the number of I/O operations to be executed, expected CPU usage, etc.
5. Expected execution time. This is usually hard to predict (see Sect. 3).
6. Data requirements — Read sets and write sets of transactions.
7. Periodicity — If a transaction is periodic, what its period is.
8. Time of occurrence of events — At what point in time will a transaction issue a read/write request?
9. Other semantics — Is the transaction read only? Does it conflict with any other transaction? If so, will they ever be executed at the same time? How up-to-date the data is required by the transaction?

There are many ways that this information can be used to help the design of real-time transactions. We demonstrate its use by the following examples.

One example concerns database consistency. In a conventional database system, as long as transaction atomicity, consistency, isolation and durability (the ACID properties) are enforced, transactions can be executed concurrently to increase throughput without jeopardizing correctness. However, insurance of the ACID properties does not come cheap. Special protocols for concurrency control, transaction commitment, and database recovery have to be exercised. Very often, such protocols hamper the system's real-time performance through blocking, transaction abortion, deadlock, and additional I/O due to logging.

Since full transaction support is costly, it has been suggested [51] that real-time data and transactions be categorized into classes depending on their timing, synchronization, consistency, and atomicity properties. Then using the supplied semantic information, devise special minimal transaction supports that are sufficient for the classes.

Another example use of semantic information as suggested in [16] is to analyze transactions and construct contingency plans for each transaction type. Contingency plans are alternate actions that can be invoked whenever the system determines that it cannot complete a task in time. A contingency plan is usually more economical to execute than the original transaction. It provides useful but not optimal results. A related idea on imprecise computations can be found in [14].

Among the attributes of a real-time transaction, deadline is the most important one. This piece of information is used in many aspects of a RTDBS, be it concurrency control, scheduling, or the use of contingency plans and imprecise computations. Usually the deadline of a transaction is specified by the application designers. However, if the transaction model supports nested transactions or subtransactions, there is the question of how time constraints are assigned to individual subtransaction based on the parent transaction's deadline.

To illustrate this problem, let's consider a transaction T with deadline d. Further assume that T consists of two subtransactions T_1 and T_2 to be executed in order. Since T_2 is executed last, its deadline should be d, the deadline of its parent transaction. But what about T_1's deadline? If we set it to be d minus the expected execution time of T_2, T_2 is left with no slack[4] and the system runs the risk of missing T's deadline. A probably better but more complicated solution is to assign a tighter deadline to T_1. If T_1 misses it, its deadline is incremented gradually until it is completed. A problem with this scheme is that transactions with "soft" but tight deadlines (e.g. T_1) will interfere with the execution of others that have "harder" deadlines (e.g. T_2).

So far we have assumed that real-time constraints are specified on the transactions. Korth et. al. [32] propose a different model with which deadlines are associated with consistency constraints. In addition to transactions that maintain correct database states, in their model, transactions may be invoked to record the effects of some external event that is generated outside the system (e.g. sensor reading). The ensuing change in the database state may render a consistency constraint invalid (e.g. room temperature $< x^o$F), and that constraint may need to be restored within a specific deadline (e.g. the room temperature has to be raised within 30 seconds). Once an inconsistency is detected, a "patch-up" transaction is invoked to attempt to correct the violation. The patch-up transaction, however, may cause other consistency constraints to be violated. This leads to a possible chain of transaction triggering.

[4] The slack time of a transaction is the amount of time that the transaction can be delayed in its execution but still be able to meet its deadline. We will have a more precise definition of slack time later in this chapter.

In [32], three types of transactions, which have *different atomicity and consistency requirements*, are considered:

1. External-input transactions: These transactions are executed to record relevant events that occur in the external world into the database. They are often simple, write-only transactions with short duration. In order to keep the database externally consistent, external-input transactions should be able to execute promptly without waiting or blocking. They may cause a consistency violation.
2. Internal transactions: These transactions are similar to standard database transactions. They are also invoked to restore consistency of the database. Their execution could be of long-duration.
3. External-output transactions: These transactions cause actions to be performed in the external world. Just like the external-input transactions, they are often of short-duration.

Their approach to consistency restoration works as follows: First of all, by analyzing the underlying real-time system, a predicate-priority graph (PPG) is constructed. A PPG is a bipartite graph consisting of two kinds of nodes representing transactions and consistency constraints. An edge emerging from a transaction node, T_1, to a constraint node, C_1, means that the execution of T_1 may cause C_1 to be violated. The fanout of a transaction node may be larger than one, meaning that a transaction can potentially violate several consistency constraints. An edge from a constraint node, C_2, to a transaction node T_2 symbolizes that by executing T_2, C_2 will be restored. Again, a constraint node may have multiple outgoing links. In that case, *any one* of the transaction nodes that are pointed to by a constraint node is capable of restoring the constraint. A choice is thus possible in selecting a "patch-up" transaction.

Now, when a constraint is violated, a "patch-up" plan is constructed by analyzing the PPG. A "patch-up" plan is represented by an inconsistency-resolution subgraph (IRS) of the PPG, which provides a strategy for resolving any inconsistencies. Intuitively, an IRS gives a partial ordering of transaction execution so that consistency constraints are restored.

Since a constraint violation may be fixed by more than one internal transaction, there may be more than one IRS choice for restoring an inconsistency. Korth's paper suggests several strategies for selecting an IRS. For example, choose an IRS such that:

1. the total execution time of the transactions involved is minimum.
2. the IRS involves the least number of transactions.
3. the IRS violates the least number of consistency constraints.
4. the slack time for restoring consistencies is maximum.

These strategies are engineered towards different system performance metrics. Complexities of problems related to the implementation of these strategies are also studied in [32]. Some of these problems are found to be NP-hard.

3 Transaction Scheduling

A major part of real-time system research concerns scheduling of jobs (of which transactions are one kind) in a multiprogramming environment. Following Liu and Layland's paper [39], numerous others have been published on the subject. Among these is a series of work done by Lehoczky, Sha et. al. [34] [35] [36]. For a survey on scheduling algorithms in a hard real-time environment readers are referred to [13].

Much of the work done on real-time job scheduling focuses mainly on CPU scheduling. Transaction scheduling, however, involves not only the CPU. In fact, due to the extensive data processing requirements of a database system, resources such as data, I/O, and memory are also subject to severe competition among concurrently running transactions. Careful scheduling the use of these resources is very important to the performance of RTDBSs.

In this section, we discuss some general issues of transaction scheduling. Since most of the real-time scheduling protocols revolve around the use of priority, we will discuss how priority is assigned to transactions. We also discuss CPU scheduling and its database related problems. Algorithms for scheduling other system resources such as data, I/O, and memory will be discussed in the following sections on concurrency control, I/O scheduling, and buffer management respectively.

As a major asset of a computer system, efficient use of CPU cycles is very important. Conventional scheduling algorithms [43], as employed by most of the existing operating systems, aim at balancing the number of CPU-bound and I/O-bound jobs to maximize system utilization and throughput. They are also designed to treat processes fairly, each one gets its fair share of the system resource. Other performance criteria include small job turnaround time, small waiting time, and fast response time. However elaborated, these algorithms are not adequate for real-time transaction scheduling. This is because in a RTDBS, transactions should be scheduled according to their criticalness and the tightness of their deadlines, even if this means sacrificing fairness and system throughput.

Real-time scheduling algorithms should therefore be based on the "inequalities" of transactions. They should give preferential treatment to transactions which are very critical and with stringent timing constraints. A popular method is to assign a numeric priority to each transaction which reflects its relative urgency. A transaction with higher priority is given an upper hand in gaining access to system resources.

A transaction has many attributes that may affect its priority. Below is a list of those attributes that are most relevant to a RTDBS. The parenthesized variables next to each attribute represent the individual quantitative measure of each concept.

1. Criticalness (γ) — the more critical a transaction is, the higher is its priority.[5]

[5] Sometimes, the criticalness of a transaction can be expressed as a value function (see Sect. 2). For scheduling algorithms that aim at obtaining high total process value, see [40] [23].

2. Deadline (d) — the earlier its deadline, the higher is the transaction's priority [22].
3. Amount of unfinished work (l) — a transaction with less amount of unfinished work may be given a higher priority than a transaction with large amount of unfinished work. In the extreme case when a transaction has begun its commit phase[6], its priority could be raised to a higher value. This enables a committing transaction, who requires minimal computation, to finish fast. Resources held by the committing transaction can thus be released sooner to reduce blocking of other transactions [24].
4. Amount of computation already invested (c) — a transaction that already has a large amount of computation done may be given a higher priority. Preempting a transaction in a database system requires not only the release of resource but also careful rollback of the transaction. It is sometime easier and less wasteful of system resources to rollback a transaction that has only run for a short time.
5. Age (a) — a transaction that arrived early should be given a higher priority than those that arrived late. This scheme reduces turnaround time and helps keep data externally consistent.
6. Slackness (s) — slackness measures how long a transaction's execution can be delayed while still making it possible to meet the transaction's deadline. If we denote the arrival time of a transaction by t_a, then slackness can be expressed as:
$$s = d - t_a - c - l.$$
The tighter the slackness of a transaction is, the higher should be its priority.

It is generally hard to capture the idea of urgency by only one of the items discussed above. Consequently, it is suggested in [51] that a combination be used to compute a priority value function ($pr()$). In particular, the following formula is suggested as an example:

$$pr(T) = \gamma(w_1 a - w_2 d + w_3 c - w_4 l)$$

where the $w_i's$ are weights reflecting the relative importance of the various factors.

We note that when priority computation is based on the amount of unfinished work and slackness, a good prediction of transaction execution time is needed. We have discussed in Sect. 1 the factors which make a precise prediction hard to achieve. As an attempt, we can generally decompose the execution time (t_{exec}) into three components as follows [9]:

$$t_{exec} = t_{fault} + t_{db} + t_{nondb}$$

where t_{fault}, t_{db}, and t_{nondb}[7] denote the times spent in page fault, data-processing operations, and non-data-processing operations respectively. We look at these terms one by one.

[6] A transaction is in its commit phase after it finishes all the computation. Any data it updates are being written to disk in this phase.

[7] This notation is adapted from [9].

The term t_{fault} represents the amount of time spent in paging data from disk to memory. For periodic transactions, if data prefetching is possible, a memory resident database can be assumed. This removes any uncertainty on t_{fault} by essentially setting it to zero. Otherwise, t_{fault} includes all the time for I/O operations. Due to the wide gap between memory access time and disk access time, in a disk-based database, the use of a deterministic worst-case bound on t_{fault} is too pessimistic. A probabilistic model on estimating t_{fault} may be more effective in this case. Scheduling algorithms which are based on execution time prediction, therefore, have to take into account the fact that the estimates are not precise.

The variable t_{nondb} measures the execution time of non-database related operations while t_{db} measures database related ones. It is generally harder to estimate t_{db} than t_{nondb}. The reason being that the amount of data processing usually depends on the state of the database itself. It is suggested that metadata be kept describing the size of each object class [9]. Execution time on data processing is then estimated dynamically with the help of these metadata.

Before we end this section, we briefly discuss various scheduler properties and compare their relative merits with respect to RTDBSs. These properties include on-line vs. off-line, conflict-avoidance vs. conflict-resolution, and preemptive resume schedulings.

Due to the unpredictable job arrival pattern, conventional scheduling algorithms are usually on-line. That is, the order of transaction execution is not pre-computed. However, in RTDBS, if information about the transactions' data access patterns, periodicities, deadlines etc. is available, transaction preanalysis should be carried out off-line [9]. Transaction execution order is thus scheduled before transactions arrive. Since off-line schedulers are given more information, and sooner, they are more flexible and usually produce better schedules.

When there are concurrently running tasks in a system, there are potential conflicts on resource access. These resources include data, I/O, memory and others. When given a job, a conflict-avoidance scheduler detects and resolves conflicts among jobs over resources before the job is released for execution [9]. For conflict-avoidance be applicable, all resource requirements must be known in advance. A conflict-resolving scheduler, on the other hand, handles conflicts when they actually occur. A conflict-resolution protocol, for example, may decide that a resource requester aborts a resource holder, if it is determined that the requester has a higher priority over the resource. The penalty of using a run-time conflict resolution strategy is the uncertainty it introduces in transaction execution time [9].

Finally, we note that preemptive resume CPU scheduling may not be suitable for database systems [11]. Under this scheme, a high priority transaction preempts a low priority one for CPU. The low priority transaction is not aborted and does not relinquish any lock held. It simply sleeps and then resumes processing when the high priority transaction completes. If the low priority transaction is holding lock on a hot item, a convoy of waiting transactions will be formed due to the extended period of locking. This convoy, once formed, tends to persist for a long time [8]. The convoy phenomenon causes long waits for locks and should be avoided in a real-time system. A solution based on priority-based round-robin CPU scheduling is suggested in [11], where the length of a CPU slice is determined by the priority of a transaction.

4 Concurrency Control

Concurrency control refers to the control of interaction among concurrent transactions in such a way that database consistency is not destroyed [31]. Transactions interact with each other mainly through reads and writes of data items. Careful access control on data therefore needs to be exercised. A good deal of work has been done on this subject for conventional databases (see, for example, [42]). The purpose of this section is to discuss the properties of concurrency control protocols that are pertinent to RTDBSs.

Serializability is the most popular correctness criterion in concurrency control. A sequence of database operations is considered serializable if its effect is *equivalent* to a serial transaction schedule. This condition, however, often limits the degree of multiprogramming, and introduces blockings and restarts of transactions.

An argument which supports sacrificing serializability to improve performance in a RTDBS is that data are often short-lived in some real-time applications [48]. The claim is that any inconsistency introduced by concurrent transactions does not spread too much over the database. Since the content of the database does not get corrupted badly, techniques like compensating transactions as discussed in Sect. 2 may be useful.

However, depending on the application semantics, serializability may be a better choice for maintaining database consistency. In this case, the prevalent approaches to concurrency control are lock-based protocols and optimistic concurrency control protocols.

Two phase locking (2PL) is the most common locking protocol in conventional database systems. With 2PL, a transaction execution consists of two phases. In the first phase, locks are acquired but may not be released. In the second phase, locks are released but new locks may not be acquired. In case a transaction T_R requests a lock that is being held by another transaction T_H, T_R waits.

Conventional locking protocols, like 2PL, are unsatisfactory for RTDBSs. The two main problems encountered are the possibility of priority inversion and deadlock. Let's take a look of the problem of priority inversion first.

Consider the example given above which involves a lock requester T_R and a lock holder T_H. If the priority of T_R is higher than the priority of T_H, then a high priority transaction waits for a low priority one to finish. We call this phenomenon *priority inversion* [2], [4], [45], [27].

Priority inversion is very undesirable in a RTDBS because a high priority transaction is blocked by a low priority one. Since the low priority transaction is discriminated against in its use of system resources, the blocked high priority transaction is essentially running at an effective priority equal to that of the low priority transaction. This renders the real-time scheduling algorithms ineffective.

One solution to this problem is to hoist the priority of the lock holder to that of the requester. Referring to our earlier example, T_H will be executed at an elevated priority equal to $pr(T_R)$. This priority lift truly reflects the urgency of completing T_H, whose progress means progress of T_R. We call this strategy *Wait Promote* [45].

Wait Promote:
IF $pr(T_R) > pr(T_H)$ **THEN**
  T_R waits;
  T_H inherits the priority of T_R;
ELSE
  T_R waits;
ENDIF

We note that the property of priority inheritance, as exhibited by the Wait Promote strategy, should be transitive. It means that if T_H is itself blocked by some other transaction X, then we should set $pr(X) = \max\ \{pr(X), pr(T_R)\}$. Also, if a lock holder is blocking more than one lock requester, the priority of the lock holder should be set to the maximum of the requester's priorities.

The problem with Wait Promote is that we still let a low priority transaction block a high priority transaction. If aborting a transaction is not too expensive, we may choose to abort the low priority lock holder and let the high priority lock requester proceed. This strategy is called *High Priority* [2].

High Priority:
IF $pr(T_R) > pr(T_H)$ **THEN**
  T_R aborts T_H;
ELSE
  T_R waits;
ENDIF

The use of High Priority eliminates the problem of priority inversion. However, a problem arises if the priority function chosen (e.g. least slack) is such that a restarted transaction may have a higher priority than its previous incarnation. In such cases, when the restarted transaction tries to acquire locks, it may abort the transaction that killed it before because the restarted transaction is now running at a higher priority. This leads to the problem of cyclic restart.

To avoid this problem, before a lock requester T_R aborts a lock holder T_H, the scheduler should make sure that the next incarnation of T_H, T_H^A, also has a lower priority than T_R. This modified *High Priority* algorithm *without cyclic restart* is shown below [2]:

High Priority without Cyclic Restart:
IF $pr(T_R) > pr(T_H)$ **AND** $pr(T_R) > pr(T_H^A)$ **THEN**
  T_R aborts T_H;
ELSE
  T_R waits;
ENDIF

The High Priority strategy, although simple, may abort transactions too liberally. This wastes system resource and lower throughput, and should be avoided unless it is necessary. For our example, if it is estimated that the slack time of T_R is longer than the remaining running time of T_H, then T_H may be allowed to finish without missing T_R's deadline. In that case, T_H is not aborted to save system resource. This strategy, called *Conditional Restart* [2], is shown below:

Conditional Restart:
E_H := estimated remaining running time of T_H;
S_R := estimated slack time of T_R;
IF $pr(T_R) > pr(T_H)$ **AND** $pr(T_R) > pr(T_H^A)$ **THEN**
    IF $S_R \geq E_H$ **THEN**
        T_R waits;
        T_H inherits the priority of T_R;
    ELSE
        aborts T_H;
    ENDIF
ELSE
    T_R waits;
ENDIF

There are two complications of Conditional Restart. First, if there is a non-trivial probability that the chain of blocked transactions involves more than one transaction, the strategy needs to be modified. For example, if T_H is itself blocked by a transaction X, then instead of comparing E_H and S_R, we ought to compare the sum of the expected execution times of H and X with S_R instead. Second, estimates of E_H and S_R have to be available.

The above discussion shows that no single strategy excels. The choice is dependent upon the applications, the availability of resource, and the cost of transaction restart.

There are studies on other real-time locking protocols which use delayed transaction commitments to achieve more flexible schedulings, and to produce serialization orders that favor high priority transactions [49] [38] [6]. In [49], a three-phase real-time locking protocol is proposed. Among its nice properties are strict history [7], high degree of concurrency, and a smaller rate of missed deadlines compared to the basic 2PL-HP protocol. In [6], the problem of blocking and restarts caused by conventional locking protocols, and their adverse effects on a RTDB are discussed. A technique called "ordered sharing" [5] is used to tackle the problem and is shown to be an effective way of ameliorating blockings and restarts, as well as transaction missed deadlines.

As mentioned earlier, the second problem of locking protocols is the possibility of deadlock. Whenever a set of transactions get involved in a circular wait, a deadlock occurs [43]. In such situation, a transaction involved in the deadlock is chosen to be aborted. This victim transaction should be picked such that the largest number of remaining transactions can meet their deadlines. Example strategies for choosing a victim in deadlock resolution include [24]:

1. Abort a transaction that already passed its deadline.

2. Abort a transaction with the longest deadline.
3. Abort a transaction that is least critical.

Finally, empirical studies have shown that when deadlock occurs, it usually involves only two transactions [17]. There are thus not many choices for a victim. Hence, it may not be wise to use a sophisticated but expensive deadlock breaking protocol.

Most commercially available database systems use lock-based concurrency control protocols. Optimistic concurrency control, however, has the advantages of being non-blocking and deadlock free. These properties are very desirable for a real-time system. We devote the rest of this section to a discussion of optimistic concurrency control as applied to RTDBSs [21], [26], [46].

With optimistic concurrency control, the execution of a transaction can generally be divided into three phases: (1) read phase, (2) validation phase, and (3) write phase.

During the read phase, data items are read into memory. Computations based on the values of these data items are performed. New values are computed, but are not written into the database until the write phase. In general, if the concurrency control scheduler has decided that a transaction T_i be serialized before a transaction T_j, the following conditions have to be satisfied [26]:

1. R/W rule. Data items to be written by T_i should not have already been read by T_j.
2. W/W rule. T_i's writes should not overwrite T_j's writes.

When a transaction finishes its computation, it enters its validation phase in which the R/W and W/W rules are tested. If any one of the rules is violated, conflict resolution, which usually involves aborting one or more transactions, is invoked. One scheme for validating the rules is to check if any one of the following condition hold:

1. T_i completes its execution before T_j started (no interleaving).
2. The write set of T_i does not intersect with the read set of T_j (thus enforcing the R/W rule), and T_i completes its write phase before T_j starts its validation phase (this enforces the W/W rule).

Readers are referred to [31] for details on this validation scheme.

When validation fails, a conflict resolution scheme is invoked. Several schemes are suggested in [26]. We quote three examples here:

1. Broadcasting Commit. Always let the validating transaction commit and abort all the conflicting transactions. This strategy guarantees that as long as a transaction reaches its validation phase, it will always finish.
2. Abort the validating transaction only if its priority is less than that of all the conflicting transactions.
3. If the priority of the validating transaction is not the highest among the conflicting transactions, wait for the conflicting transactions with higher priority to complete.

Simulation experiments have been carried out in [20] comparing 2PL with High Priority and optimistic concurrency control with Broadcasting Commit. Their results show that under an overload management policy of discarding tardy transaction, optimistic concurrency control can outperform 2PL. An independent study by Huang and Stankovic [26] also compares an optimistic concurrency control algorithm (OCCL_SVW) with 2PL. Their results show that the performance difference between OCCL_SVW and 2PL is sensitive to the amount of data contention, but not to the amount of I/O resource contention. In particular, the optimistic concurrency control protocol performs better than 2PL when data contention is low; otherwise, 2PL has a better performance.

5 I/O Scheduling

In a disk-based database system, disk I/O occupies a major portion of transaction execution time. As with CPU scheduling, disk scheduling algorithms that take into account timing constraints can significantly improve the real-time performance [11] [3] [12]. CPU scheduling algorithms, like Earliest Deadline First and Highest Priority First, are attractive candidates but have to be modified before they can be applied to I/O scheduling. The main reason is that disk seek time, which accounts for a very significant fraction of disk access latency, depends on the disk head movement. The order in which I/O requests are serviced, therefore, has an immense impact on the response time and throughput of the I/O subsystem. To illustrate, let's consider the following example as shown in Fig. 2.

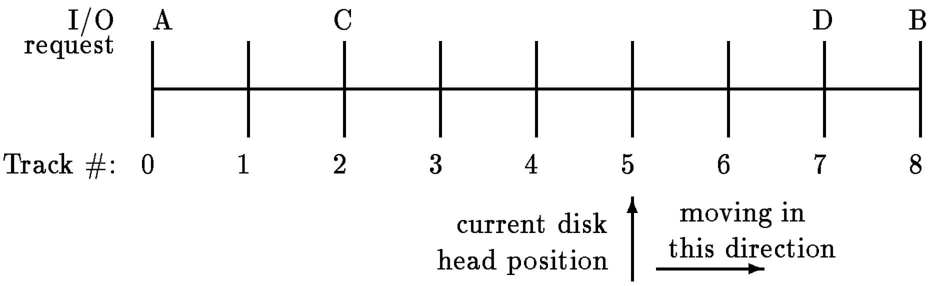

Fig. 2. Disk scheduling example.

Suppose we have four requests A, B, C and D in the I/O queue with their priorities in the following order:

$$pr(A) > pr(B) > pr(C) > pr(D).$$

The position of the data needed by each request is shown in Fig. 2. If Highest Priority First (HPF) scheduling is employed, the service order would be:

$$\text{HPF:} \quad A,\ B,\ C,\ D.$$

We note that in this case, the head sweeps the disk back and forth four times, or 32 tracks. Considering that the requests can be satisfied in only 11 track movement (in the order of D, B, C, A), apparently HPF is not a very smart way of scheduling the disk head if response time or throughput is a concern.

Algorithms for shortening disk head movement have been devised [53]. The Elevator Algorithm, for example, moves the head from one end of the disk to the other and then back, servicing whatever requests are on its way, and changing direction whenever there are no more requests ahead in its direction. Referring to the example in Fig. 2, the Elevator Algorithm will produce the following servicing schedule:

$$\text{Elevator:} \qquad D,\ B,\ C,\ A$$

which takes three times less disk head movement than Highest Priority First does.

The problem with the Elevator Algorithm, as applied to real-time systems, is that the priority of requests is not considered. In our example, the highest priority request A is serviced last. There is thus a trade-off between maximizing throughput and meeting system's timing constraints. Methods that combine the properties of HPF and the Elevator Algorithm are very desirable. In what follows, we describe two middle-ground I/O scheduling algorithms: one that puts the Elevator concept on Highest Priority First scheduling, and another which adds the flavor of HPF to the Elevator Algorithm.

When Highest Priority First scheduling is used, the disk head may pass through tracks for which there are other low priority requests. The Elevator principle says "do pick them up because the disk head is already there!" In [3], [4], Abbott presents the FD-SCAN[8] algorithm. Simply stated, FD-SCAN follows HPF in always "targetting" the disk head towards the track with the highest priority request, but also services whatever requests are on its way. Consider the earlier example, the servicing order under FD-SCAN would be:

$$\text{FD-SCAN:} \qquad C,\ A,\ D,\ B.$$

We note that in this example, the disk head moves a similar distance as the Elevator Algorithm but the highest priority request A is served sooner.

In Abbott's studies, FD-SCAN is tested against other disk scheduling algorithms including First Come First Served, the Elevator Algorithm, Shortest Seek Time First, and Earliest Deadline First. Simulation results show that FD-SCAN performs best among the algorithms tested in terms of the ability to meet deadlines. This property is most prominent when the load of the I/O subsystem is high. Also, this advantage of FD-SCAN is persistent through a wide range of system parameter settings.

In [11], the problem of long seek time for the Highest Priority First scheduling is addressed. It is argued that the use of fine grain priority gives the HPF scheduler a FCFS-like average seek time (with possibly even worse response time).

[8] In [3] and [4], deadline is used as a priority measure. FD-SCAN stands for "Feasible Deadline SCAN." Any request whose deadline is determined to be impossible to meet is discarded.

Their idea (which we will call the Highest Priority Group First (HPGF)) is to blur the boundaries of priority. Disk requests are grouped into a small number of priority levels even though the transactions issuing the I/O requests may have distinct priorities in other parts of the system. Once these groups are formed, the disk is scheduled to service the highest priority group first. In case there is more than one request in the highest priority group, the Elevator Algorithm is used for the intra-group scheduling. Referring to our example, if requests A and B are in a high priority group, and requests C and D are in a low priority group, the service order under HPGF would be:

$$\text{HPGF:} \qquad B,\ A,\ C,\ D.$$

We note that in the example, the disk movement is much less than what HPF would require, while the higher priority requests are served before the lower priority ones.

Through a series of experimental studies [11], it is found that HPGF performs better than the Elevator Algorithm in meeting deadlines. This benefit is achieved at a cost of a prolonged *average* response time. However, the study shows that the response time degradation mainly affects low priority requests. High priority requests, on the other hand, experience response times which are very close to what the Elevator Algorithm provides.

6 Buffer Management and Memory Resident Database

In the last three sections, we have discussed various issues concerning access to CPU, data, and disk I/O in a real-time database system. In this section, we turn our attention to yet another system resource — main memory. We will discuss how memory is managed and how it can be used efficiently to improve the performance of RTDBSs.

This section is divided into two parts. The dividing issue is whether memory space is tight or plentiful. If a real-time system has only limited amount of memory, *buffer management*, which concerns the allocation of memory space among concurrent transactions, has to be specially designed. The goal here is to ensure that the execution of high priority transactions is not hindered by the lack of memory. On the other hand, if memory is plentiful, much of the data can reside in main store[9] forming what is called a *memory resident database system* (MRDBS). An MRDBS has many features, such as fast and predictable access time, which make it particularly suitable for real-time applications.

6.1 Buffer Management

The availability of memory affects transaction response time in two ways. First, before a transaction starts its execution, buffers (memory pages) have to be allocated to the transaction. These buffers are used to store the execution code,

[9] We use the word "store" as a synonym of "memory".

copies of files and data paged in from disk, and any temporary objects produced. Depending on the transaction, a certain number of buffers have to be allocated in order to prevent the transaction from *thrashing*[10]. When memory is running low, a transaction may be blocked from execution. The amount of memory available in a system thus limits the number of concurrently executable transactions. Second, some applications, such as image processing, have high demands on memory. Their executions will be significantly slowed down if memory is tight and frequent memory swapping is done.

The job of a buffer manager is to allocate memory buffers to transactions intelligently such that high priority transactions enjoy shorter response times. A buffer manager is usually specified in terms of its admission policy and buffer replacement policy. We briefly explain each policy in turn below. We will also give examples on how transaction priorities are used to improve the manager's real-time behavior [25].

When a transaction T is issued, the buffer manager will decide whether to admit it for execution. This decision is called the transaction admission policy. We assume that transactions be able to supply the buffer manager with the number of buffers it needs for proper execution. If enough free space is available, transaction T is admitted. Otherwise, T is blocked or else a number of running transactions can be suspended[11] and their buffers reallocated to transaction T. For the latter case, the decision of which transactions to suspend can be determined by their priorities. A simple solution would be to suspend transactions with the lowest priorities until either:

1. enough number of buffers have been freed up for T to execute, or
2. there are no more unsuspended transactions with priority less than that of T.

In the first case, the freed-up buffers are allocated to T and T is admitted. For the second case, T is blocked due to a lack of memory.

When a transaction references a data item which is not already in memory, a free buffer has to be allocated to page in the data. If no more free buffers are available, some buffer has to be flushed out to disk (if it was dirty) and its content replaced by the needed data. The choice of a buffer for replacement is called the buffer replacement policy.

Traditional replacement policies include Least Recently Used (LRU), Least Frequently Used etc. [43]. In [11], a new policy, Priority-LRU, is proposed which considers transaction priority as well as buffer recency. This algorithm groups transactions into m priority classes. All buffers which are being used by some transaction in the i^{th} class is put into a list L_i and are said to be of priority i. The buffer pool is thus organized into m lists: L_1, L_2, ..., L_m according to buffer priority. The Priority-LRU algorithm can be succinctly described by the following pseudo code:

[10] In our context, thrashing refers to the phenomenon in which a transaction spends most of its time swapping data to and from disk [43].

[11] A suspended transaction is swapped out to disk and its execution is halted.

```
Priority-LRU(W_R):
S := ϕ;
(* put the least recently used buffer of each list into S *)
FOR i := 1 TO m DO
      x := least recently used buffer in L_i;
      S := S ∪ {x};
END FOR
(* pick the lowest priority buffer in S that is not one of the W_R
   most recently used buffers *)
WHILE S ≠ ϕ DO
      x := lowest priority buffer in S;
      (* test if x has been referenced recently *)
      IF x is one of the W_R most recently accessed buffers THEN
         S := S − {x};
      ELSE
         RETURN(x);
      ENDIF
END WHILE
RETURN(no suitable page);
```

The Priority-LRU algorithm takes one parameter, W_R, which controls the relative importance of recency and priority. For example, when W_R is set to zero, the least recently used buffer in the lowest priority group is chosen. A low priority buffer is always chosen in favor of higher priority ones. Conversely, if W_R is set high, then low priority buffers will get a break if they are referenced recently enough.

6.2 Memory Resident Database System

As discussed in Sects. 1 and 3, one of the major difficulties encountered in designing a RTDBS is the long and often unpredictable disk access delays. As the price of memory continues to drop, one possible remedy is to put data directly into memory, thus eliminating I/O accesses. In this subsection, we give a brief account on memory resident database system design. Interested readers are referred to [10], [18], [19] and [48] for further reading.

Compared to disk, main memory access time is much faster (1000 – 10000 times), and is more predictable (no disk seek). These features are very desirable in RTDBSs, and may even be necessary if transactions have extremely tight time constraints.

However, putting all the data in memory is not without its disadvantages. Above all, an MRDBS is more costly than a disk-based system. Even though technology for high density memory chips is improving and the cost dropping, currently there is still a limit on how much data can be memory resident. For large databases, storing data in main memory has to be done selectively. In a real-time environment, if transaction data requirement is relatively stable and known, data items that are referenced by high priority transactions should have preference over low priority ones in claiming memory residency.

Another problem with main memory is its volatility. Data stored in main memory usually do not survive through a power failure, nor a CPU failure. An MRDBS, therefore, still requires disks to provide a stable backup storage. Conventional recovery protocols that load the entire database to memory from the disk backup copy, and then apply the transaction log to bring the database up-to-date may be too slow for real-time applications. Mechanisms which allow quick restart and the database to function (partially) during recovery have to be employed [28]. For example, in [18], a recovery technique for MRDBS is proposed. Their method assumes that a small part of main memory is made stable by separate battery backup. This stable memory is used to store log records of "pre-committed" transactions. Schemes for check-pointing the database and compressing the transaction log for fast restart are also discussed.

A third MRDBS issue is that their design goals are different from a conventional disk-based system. Data structure and query processing algorithms for traditional database system are optimized to reduce the number of disk accesses and to enhance data clustering [31], [56]. These goals are no longer valid[12] in an MRDBS. When data are memory resident, query optimization and data structure should be designed to minimize CPU processing time and the amount of memory space used. Conventional access methods and database structures have to be revised. A B-tree [15], for example, is found to be less efficient than hashing for MRDBS index search. This is due to the additional space B-tree needs to store all the keys and pointers [10]. The sort-merge join algorithm [30], which was designed to reduce the number of disk I/O, is also found to be inferior in performance than the hash-merge algorithm when memory is plentiful [10].

Finally, small data access time also affects the choice of a concurrency control mechanism [48]. Without I/O delay, transaction execution time will be small in an MRDBS. Blocking delays due to data locking will also be reduced. We can thus afford to have a coarser granularity for data lock to reduce memory and processor overhead. Moreover, since memory is an important asset, optimistic concurrency controls that create temporary data objects, and those which store multiple versions of data, may not be attractive in an MRDBS [48].

7 Conclusion

In this chapter, we have discussed the various issues concerning the design and implementation of real-time databases and transaction processing. We distinguished a RTDBS from a database system and a real-time system by its more demanding goals. We also looked at application semantics and showed how they can be used to improve RTDBSs performance. CPU, data, I/O, and memory

[12] Clustering may still improve data access time in an MRDBS by putting data that are often referenced together in the same "cache line". This increases the cache hit probability. The impact is, however, not as dramatic as data clustering on disk.

scheduling were also discussed. Furthermore, some desirable features of memory resident databases as applied to a real-time environment were also mentioned.

Due to space limitation, some other aspects of RTDBS which deserve special attention are not covered by this chapter (see [44], [54] for additional discussion). These topics include fast and incremental recovery protocols [28], database architectures that support predictable transaction execution, programming languages that provide constructs for timing specifications [55], query processing and optimization techniques that are based on real-time performance goals, scheduling methods that improve data external and temporal consistency [50] and distributed real-time databases [37].

Finally, we note that appropriate deadline assignment to subtransactions is very crucial to the success of many real-time database protocols [29] [41]. Relatively little work has been done on this subject. As real-time databases evolve, however, we expect to see more work on this, and many other RTDBS problems.

References

1. R. Abbott, H. Garcia-Molina: What is a Real-Time Database System? Abstracts of the Fourth Workshop on Real-Time Operating systems, IEEE (July 1987) 134–138
2. R. Abbott, H. Garcia-Molina: Scheduling Real-Time Transactions: a Performance Evaluation. Proceedings of the 14th VLDB Conference (August 1988) 1–12
3. R. Abbott, H. Garcia-Molina: Scheduling I/O Requests with Deadlines: a Performance Evaluation. IEEE Real-Time System Symposium. (Dec. 1990) 113–124
4. R. Abbott: Scheduling Real-Time Transactions: a Performance Evaluation. Ph.D. Dissertation, Princeton University. (1991)
5. D. Agrawal, A. El Abbadi, A. E. Lang: Performance Characteristics of Protocols with Ordered Shared Locks. Proceedings of the 18th IEEE Internatinal Conference on Data Engineering. (1991)
6. D. Agrawal, A. El Abbadi R. Jeffers: Using Delayed Commitment in Locking Protocols for Real-Time Databases. Proceedings of the ACM SIGMOD International Conference on Management of Data. (1992) 104–113
7. P. A. Bernstein, V. Hadzilacos, N. Goodman: Concurrency Control and Recovery in Database Systems. Addison-Wesley. (1987)
8. M. Blasgen, J. Gray, M. Mitoma, T. Price: The Convoy Phenomenon. Operating Systems Review. **13**(2) (1979) 20–25
9. A. P. Buchmann, D. R. McCarthy, M. Hsu, U. Dayal: Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control. IEEE (1989)
10. M. J. Carey, T. J. Lehman: Query Processing in Main Memory Database Management Systems. Proceedings of ACM SIGMOD. (1986) 239–250
11. M. J. Carey, R. Jauhari, M. Livny: Priority in DBMS Resource Scheduling. Proceedings of the 15th VLDB conference (1989) 397–410
12. S. Chen, J. A. Stankovic, J. F. Kurose, D. Towsley: Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems. Real-Time Systems Journal (1991) Vol. 3, number 3 307–336

13. S. C. Cheng, J. A. Stankovic, K. Ramamritham: Scheduling Algorithms for Hard Real-Time Systems — A Brief Survey. Hard Real-Time Systems, IEEE (1988) 150–173
14. J. Chung, J. Liu, W. Shih: Fast Algorithms for Scheduling Imprecise Computations. IEEE Real-Time System Symposium. (Dec. 1989) 12–19
15. D. Comer: The Ubiquitous B-Tree. ACM Computing Surveys **11(2)** (June 1979)
16. U. Dayal et. al.: The HiPAC Project: Combining Active Databases and Timing Constraints. SIGMOD Record **17(1)** (March 1988) 51–70
17. C. Devor, C. R. Carlson: Structural Locking Mechanisms and Their Effect on Database Management System Performance. Information Systems **7(4)** (1982) 345–358
18. D. Dewitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, D. Wood: Implementation Techniques for Main Memory Database Systems. Proceedings of ACM SIGMOD. (1984) 1–8
19. H. Garcia-Molina, K. Salem: System M: A Transaction Processing Testbed for Memory Resident Data. IEEE Transactions on Knowledge and Data Engineering. (March 1990) Vol. 2, number 1 161–172
20. J. Haritsa, M. Carey, M. Livny: On Being Optimistic about Real-Time Constraints. Proceedings of the 9th ACM symposium on Principles of Database Systems (April 1990)
21. J. Haritsa, M. Carey, M. Livny: Dynamic Real-Time Optimistic Concurrency Control. IEEE Real-Time Systems Symposium (Dec. 1990) 94–103
22. J. R. Haritsa, M. Livny, M. J. Carey: Earliest Deadline Scheduling for Real-Time Database Systems. IEEE Real-Time System Symposium. (1991) 232–242
23. J. R. Haritsa, M. J. Carey, M. Livny: Value-Based Scheduling in Real-Time Database Systems. VLDB Journal (April, 1993) Vol. 2 Number 2 117–152
24. J. Huang, J. Stankovic, D. Towsley, K. Ramamritham: Real-Time Transaction Processing: Design, Implementation and Performance Evaluation. University of Massachusetts COINS TR 90-43 (May, 1990)
25. J. Huang, J. Stankovic: Buffer Management in Real-Time Databases. University of Massachusetts COINS TR 90-65 (July 1990)
26. J. Huang, J. A. Stankovic: Experimental Evaluation of Real-Time Concurrency Control Schemes. Proceedings of the 17th VLDB Conference (Sept. 1991) 35–46
27. J. Huang, J. Stankovic: On Using Priority Inheritance In Real-Time Databases. IEEE Real-Time Systems Symposium (Dec. 1991) 210–221
28. B. Iyer, P. Yu, Y. Lee: Analysis of Recovery Protocols in Distributed On-line Transaction Processing Systems. IEEE Real-Time Systems Symposium (Dec 1986) 226–233
29. B. Kao, H. Garcia-Molina: Deadline Assignment in a Distributed Real-Time System. Proceedings of the 13th International Conference on Distributed Computing Systems. (1993) 428–437
30. D. Knuth: The Art of Computer Programming: Sorting and Searching. Addison-Wesley (1973)
31. H. F. Korth, A. Silberschatz: Database System Concepts. McGraw Hill (1986)
32. H. F. Korth, N. Soparkar, A. Silberschatz: Triggered Real-Time Databases with Consistency Constraints. Proceedings of the 16th VLDB Conference (1990) 71–82
33. T-W. Kuo, A. K. Mok: Application Semantics and Concurrency Control of Real-Time Data-Intensive Applications. IEEE Real-Time System Symposium. (1992) 35–45

34. J. P. Lehoczky, L. Sha, R. Rajkumar: Solutions for some practical problems in prioritized preemptive scheduling. Proceedings of IEEE Real-Time Systems Symposium (1986) 181–189
35. J. P. Lehoczky, L. Sha, J. K. Strosnider: Enhanced Aperiodic Responsiveness in Hard-real-time Environment. Proceedings of IEEE Real-Time Systems Symposium (1987) 261–270
36. J. P. Lehoczky, L. Sha, B. Sprunt: Aperiodic Task Scheduling for Hard-real-time Systems. The Journal of Real-Time-Systems (1989) 27–60
37. K. Lin, M. Lin: Enhancing Availability in Distributed Real-Time Databases. ACM SIGMOD Record (March 1988) 34–43
38. Y. Lin, S. H. Son: Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order. IEEE Real-Time System Symposium. (1990) 104–112
39. C. L. Liu, J. W. Layland: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. Journal of the ACM **20(1)** (1973) 46–61
40. E. D. Jensen, C. D. Locke, H. Tokuda: A Time-Driven Scheduling Model for Real-Time Operating Systems. IEEE Real-Time System Symposium. (1985) 112–122
41. H. Pang, M. Livny, M. J. Carey: Transaction Scheduling in Multiclass Real-Time Database Systems. IEEE Real-Time System Symposium. (1992) 23–34
42. C. H. Papdimitriou: The Theory of Database Concurrency Control. Computer Science Press (1986)
43. J. L. Peterson, A. Silberschatz: Operating System Concepts. Addison-Wesley (1985)
44. K. Ramamritham: Real-Time Databases. Distributed and Parallel Databases Journal, Kluwer Academic PUblishers. (April 1993) Vol 1, Number 2 199–226
45. L. Sha, R. Rajkumar, J. P. Lehoczky: Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transaction on Computers (1990) Vol. 39, number 9 1175–1185
46. L. Sha R. Rajkumar, J.P. Lehoczky: Concurrency Control for Distributed Real-Time Databases. ACM SIGMOD Record (March 1988) 82–98
47. L. Sha, R. Rajkumar, S. H. Son, C. Chang: A Real-Time Locking Protocol. IEEE Transactions on Computers. (July, 1991) Vol 40, Number 7 793–800
48. M. Singhal: Issues and Approaches to Design of Real-Time Database Systems. ACM SIGMOD Record (March 1988) 19–33
49. S. H. Son, S. Park, Y. Lin: An Integrated Real-Time Locking Protocol. Proceedings of the 18th IEEE Internatinal Conference on Data Engineering. (1992) 527–534
50. X. Song, J. Liu: How Well Can Data Temporal Consistency be Maintained? Proceedings of IEEE Symposium on computer-Aided Control System Design. (March 1992)
51. J. Stankovic W. Zhao: On Real-time Transactions. ACM SIGMOD Record **17** (March 1988) 4–18
52. J. A. Stankovic, K. Ramamritham: What is Predictability for Real-Time Systems? Editorial, Real-Time Systems Journal. (Nov. 1990) Volume 2, number 4 247–254
53. T. J. Teorey, T. B. Pinkerton: A Comparative Analysis of Disk Scheduling Policies. Communications of the ACM **15(3)** (March 1972) 177–184
54. Özgür Ulusoy: Current Research on Real-Time Databases. SIGMOD RECORD. (Dec. 1992) Vol. 21, Number 4 16–21
55. P. van der Stok: The Feasibility of a Relational Database Programming Language in Process Control. IEEE Real-Time Systems Symposium (Dec 1984) 105–113
56. G. Wiederhold: Database Design. McGraw-Hill (1983)

This article was processed using the LaTeX macro package with LMAMULT style