Real-Time Databases *

Krithi Ramamritham
Dept. of Computer and Information Science
University of Massachusetts
Amherst, Mass. 01003

February 28, 1996

Invited Paper – To appear in
International Journal of Distributed and Parallel Databases

Abstract

Data in real-time databases has to be logically consistent as well as temporally consistent. The latter arises from the need to preserve the temporal validity of data items that reflect the state of the environment that is being controlled by the system. Some of the timing constraints on the transactions that process real-time data come from this need. These constraints, in turn, necessitate time-cognizant transaction processing so that transactions can be processed to meet their deadlines.

This paper explores the issues in real-time database systems and presents an overview of the state of the art. After introducing the characteristics of data and transactions in real-time databases, we discuss issues that relate to the processing of time-constrained transactions. Specifically, we examine different approaches to resolving contention over data and processing resources. We also explore the problems of recovery, managing I/O, and handling overloads. Real-time databases have the potential to trade off the quality of the result of a query or a transaction for its timely processing. Quality can be measured in terms of the completeness, accuracy, currency, and consistency of the results. Several aspects of this tradeoff are also considered.

*This work was supported in part by NSF under grants CDA-8922572, IRI-9109210, and IRI-9114197.

Contents

1 Introduction

Many real-world applications involve time-constrained access to data as well as access to data that has temporal validity. For example, consider telephone switching systems, network management, program stock trading, managing automated factories, and command and control systems. More specifically, consider the following activities within these applications: looking up the "800 directory", radar tracking and recognition of objects and determining appropriate response, as well as the automatic tracking and directing of objects on a factory floor. All of these involve gathering data from the environment, processing of gathered information in the context of information acquired in the past, and providing *timely* response. Another aspect of these examples is that they involve processing both temporal data, which loses its validity after a certain interval, as well as archival data.

For instance, consider recognizing and directing objects moving along a set of conveyor belts on a factory floor. An object's features are captured by a camera to determine its type and to recognize whether it has any abnormalities. Depending on the observed features, the object is directed to the appropriate workcell. In addition, the system updates its database with information about the object. The following aspects of this example are noteworthy. First of all, features of an object must be collected while the object is still in front of the camera. The collected features apply just to the object in front of the camera, i.e., they lose their validity once a different object enters the system. Then the object must be recognized by matching the features against models for different objects stored in a database. This matching has to be completed in time so that the command to direct the object to the appropriate destination can be given before the object reaches the point where it must be directed onto a different conveyor belt that will carry it to its next workcell. The database update must also be completed in time so that the system's attention can move to the next object to be recognized. If, for any reason, a time-constrained actions is not completed within the time limits, alternatives may be possible. In this example, if feature extraction is not completed in time, the object could be discarded for now to be brought back in front of the camera at a later point in time. Applications such as these introduce the need for *real-time database systems.*

During the last few years, the area of real-time databases has attracted the attention of researchers in both real-time systems and database systems. The motivation of the database researchers has been to bring to bear many of the benefits of database technology to solve problems in managing the data in real-time systems. Real-time system researchers have been attracted by the opportunity real-time database systems provide to apply time-driven scheduling and resource allocation algorithms. However, as we shall see, a simple integration of concepts, mechanisms, and tools from database systems with those from real-time systems is not feasible. Even a cursory examination of the characteristics of database systems and the requirements of real-time systems will point out the various forms of "impedance mismatch" that exist between them. Our goal in this paper is to point out the special characteristics, in particular the temporal consistency requirements, of data in real-time databases, and show how these lead to the imposition of time constraints on transaction execution. Meeting these timing constraints demands new approaches to data and transaction management some of which can be derived by tailoring, adapting, and extending solutions proposed for real-time systems and database systems. Hence, as we present the issues in real-time database systems, we review recent attempts at developing possible approaches to addressing these issues.

This paper is divided into roughly three parts. The first part, corresponding to Sections 2, 3, and 4, introduces real-time database systems. Section 2 discusses the characteristics of *data* in real-time database systems while Section 3 presents the characteristics of *transactions* in real-time database systems. Many of these remind us of active databases. Hence Section 4 is devoted to an examination of the relationship between active databases and real-time databases to point out the additional features we need in active databases in order to make them suitable for use in a real-time database context.

The second part of the paper, contained in Section 5, discusses transaction processing in real-time database systems. We review recent research in this area and show the need to capitalize on, but *adapt*, current techniques from both real-time systems and database systems.

The third part of the paper, contained in Section 6, discusses a number of issues in real-time databases some of which have seen little or no research. These include techniques to trade off timeliness for quality, recovery of real-time transactions, and managing resources other than CPU and data. Section 7 summarizes the paper.

In the rest of this introduction, we examine those characteristics of databases and real-time systems that are relevant to real-time database systems. We also point out the advantages of using databases to deal with data in real-time systems.

1.1 Databases and Real-Time Systems

Traditional databases, hereafter referred to simply as databases, deal with persistent data. Transactions access this data while maintaining its consistency. Serializability is the usual correctness criterion associated with transactions. The goal of transaction and query processing approaches adopted in databases is to achieve a good throughput or response time.

In contrast, *real-time systems*, for the most part, deal with temporal data, i.e., data that becomes outdated after a certain time. Due to the temporal nature of the data and the response time requirements imposed by the environment, tasks in real-time systems possess time constraints, e.g., periods or deadlines. The resulting important difference is that the goal of real-time systems is to meet the time constraints of the activities.

One of the key points to remember here is that real-time does not just imply fast. Recall the story of the tortoise and the hare. The hare was fast but was "busy" doing the *wrong activity at the wrong time.* Even though we would like real-time systems to be faster than the tortoise, we do require them to possess its *predictability.* Also, real-time does not imply timing constraints that are in *nano*seconds or μseconds. For our purposes, real-time implies the need to handle *explicit* time constraints, that is, to use time-cognizant protocols to deal with deadlines or periodicity constraints associated with activities.

1.2 Why Real-Time *Databases*?

Databases combine several features that facilitate (1) the description of data, (2) the maintenance of correctness and integrity of the data, (3) efficient access to the data, and (4) the correct executions of query and transaction executions in spite of concurrency and failures. Specifically,

- database schemas help avoid redundancy of data as well as of its description,
- data management support, such as indexing, assists in efficient access to the data, and
- transaction support, where transactions have ACID (Atomicity, Consistency, Isolation, and Durability) properties, ensures correctness of concurrent transaction executions and ensure data integrity maintenance even in the presence of failures.

However, support for real-time database systems must take into account the following. Firstly, not all data in a real-time database are permanent; some are temporal. Secondly, *temporally-correct* serializable schedules are a subset of the serializable schedules. Thirdly, since timeliness is more important than correctness, in many situations, (approximate) correctness can be traded for timeliness. Similarly, atomicity may be relaxed. For instance, this happens with *monotonic* queries and transactions, which are the counterparts of monotonic tasks [35] in real-time systems. Furthermore, many of the extensions to serializability that have been proposed in databases are also applicable to real-time databases (See [41] for a review of these proposals). Some of these assume that isolation of transactions may not always be needed.

In spite of these differences, given the many advantages of database technology, it will be beneficial if we can make use of them for managing data found in real-time systems. In a similar vein, the advances made in real-time systems to process activities in time could be exploited to deal with time-constrained transactions in real-time database systems.

As illustrated by the examples cited at the beginning of this section, many real-time applications function in environments that are inherently distributed. Furthermore, many real-time systems employ parallel processing elements for enhanced performance. Hence parallel and distributed architectures are ubiquitous in real-time applications and hence real-time database systems must be able to function in the context of such architectures.

The above discussion indicates that while many of the techniques used in real-time systems on the one hand, and databases systems on the other hand, may be applicable to real-time database systems, many crucial differences exist which either necessitate fresh approaches to some of the problems or require adaptations of approaches used in the two areas. In the rest of the paper we will be substantiating this claim.

2 Characteristics of Data in Real-Time Database Systems

Typically, a real–time system consists of a *a controlling system* and a *controlled system.* For example, in an automated factory, the controlled system is the factory floor with its robots, assembling stations, and the assembled parts, while the controlling system is the computer and human interfaces that manage and coordinate the activities on the factory floor. Thus, the controlled system can be viewed as the *environment* with which the computer interacts.

The controlling system interacts with its environment based on the data available about the environment, say from various sensors, e.g. temperature and pressure sensors. It is imperative that the state of the environment, as perceived by the controlling system, be consistent with the actual state of the environment. Otherwise, the effects of the controlling systems' activities may be disastrous. Hence, timely monitoring of the environment as well as timely processing of the sensed information is necessary. The sensed data is processed further to derive new data. For example, the temperature and pressure information pertaining to a reaction may be used to derive the rate at which the reaction appears to be progressing. This derivation typically would depend on past temperature and pressure trends and so some of the needed information may have to be fetched from archival storage (a temporal database [46]). Based on the derived data, where the derivation may involve multiple steps, actuator commands are set. For instance, in our example, the derived reaction rate is used to determine the amount of chemicals or coolant to be added to the reaction. In general, the history of (interactions with) the environment are also logged in archival storage.

In addition to the timing constraints that arise from the need to continuously track the environment, timing correctness requirements in a real–time (database) system also arise because of the need to make data available to the controlling system for its decision-making activities. For example, if the computer controlling a robot does not command it to stop or turn on time, the robot might collide with another object on the factory floor. Needless to say, such a mishap can result in a major catastrophe.

The need to maintain consistency between the actual state of the environment and the state as reflected by the contents of the database leads to the notion of temporal consistency. Temporal consistency has two components [48, 4]:

- *Absolute consistency* – between the state of the environment and its reflection in the database.

  As mentioned earlier, this arises from the need to keep the controlling system's view of the state of the environment consistent with the actual state of the environment.

- *Relative consistency* – among the data used to derive other data.

  This arises from the need to produce the sources of derived data close to each other.

Let us define these formally. Let us denote a data item in the real-time database by

$$d : (value, avi, timestamp)$$

where d_{value} denotes the current state of d, and $d_{timestamp}$ denotes the time when the observation relating to d was made. d_{avi} denotes d's *absolute validity interval*, i.e., the length of the time interval following $d_{timestamp}$ during which d is considered to have absolute validity.

A set of data items used to derive a new data item form a *relative consistency set*. Each such set R is associated with a *relative validity interval* denoted by R_{rvi}.

Assume that $d \in R$.

d has a correct state iff

1. d_{value} is logically consistent – satisfies all integrity constraints.

2. d is temporally consistent:

   - Absolute consistency: $(current\_time - d_{timestamp}) \leq d_{avi}$.

   - Relative consistency: $\forall d' \in R, \mid d_{timestamp} - d'_{timestamp} \mid \leq R_{rvi}$.

Consider the following example: Suppose $temperature_{avi} = 5$, $pressure_{avi} = 10$, $R = \{temperature, pressure\}$, and $R_{rvi} = 2$. If $current\_time = 100$, then (a) $temperature = (347, 5, 95)$ and $pressure = (50, 10, 97)$ are temporally consistent, but (b) $temperature = (347, 5, 95)$ and $pressure = (50, 10, 92)$ are not. In (b), even though the absolute consistency requirements are met, R's relative consistency is violated.

Whereas a given *avi* can be realized by sampling the corresponding real-world parameter often enough, realizing an *rvi* may not be that straightforward. This is because, achieving a given *rvi* implies that the data items that belong to a relative consistency set have to be observed at times close to each other.

Also, achieving an *rvi* along with the *avi*'s will mean that smallest of the *avi*'s of the data items belonging to the relative consistency set will prevail. Consider the *temperature* and *pressure* example, where both of them belong to R. The transactions writing *temperature* and *pressure*, respectively, must always write them within 2 time units of each other. This will implicitly lower the *avi* of *pressure* to 5.

One way out of this predicament is to realize that relative consistency requirements result from the need to derive data from data produced within close proximity of each other. Thus meeting relative consistency requirements is necessary only when data is *used* to derive other data. So rather than reducing the *avi*'s, we need to ensure that an *rvi* is satisfied just when a transaction is executed to derive new data.

If two data items belong to multiple relative consistency sets, the smallest of the *rvi*'s will prevail. Suppose *temperature* and *pressure* also belong to relative consistency sets R' where $R'_{rvi} = 1$. Clearly, the timestamps of *temperature* and *pressure* must be within 1 time unit of each other to satisfy the relative consistency requirements of R and R'.

Another issue in this context relates to the manner in which *timestamps* of derived data are set. Clearly, there will be some correlation between these timestamps and those of the data from which new data is derived. One possibility is to assign the timestamp of d' derived from data items in R to be equal to $min_{d \in R}(d_{timestamp})$ [48]. That is, derived data is only as recent as the oldest data from which the derivation occurs. In general, however, temporal validity criteria are likely to be application dependent and so the timestamp of derived data can be stated as some function of those of the data in the corresponding R [4].

Let us pursue the relationships between *avi*'s and *rvi*'s further. Suppose data items u and v are used to derive data items x and y which in turn are used to derive z. As we saw earlier, z_{avi} is derived from x_{avi} and y_{avi}, which in turn are derived from u_{avi} and v_{avi}. Thus (z $derived\_from^*$ x) where $derived\_from^*$ is the transitive closure of the relation $derived\_from$. Given the *avi* and the *rvi* of the derived data, the $derived\_from^*$ relationship, and the function used to assign the timestamps of the derived data we can *determine* the *rvi* and *avi* of the data items they are derived from.

This discussion shows the interrelationships between the *derived_from* relationship, the manner in which timestamps are set for derived data, and the composition of the relative consistency sets. Furthermore, the observation that relative consistency is significant only when data is being derived is an additional consideration. Methodical approaches must be developed to address this problem such that the system is not overconstrained, i.e., temporal consistency requirements are not stricter than necessary. This is important since, as we will see in the next section, temporal consistency requirements translate into timing constraints on transactions, and the more restrictive the temporal consistency requirements, the tighter the time constraints, and the harder it is to satisfy them.

Before we conclude this section, it should be noted that *avi* and *rvi* may change with system dynamics, e.g., mode changes. For instance, while it is necessary to monitor temperature and pressure closely, i.e., have a small *avi*, during the early stages of a reaction, it might be appropriate to increase the *avi* once the reaction reaches a steady state.

Given that integrity constraints are typically expressed via predicates and temporal constraints can also be expressed via predicates, we have a set of predicates to be satisfied by data. Why not use *standard* integrity maintenance techniques? The answer lies in observing that while not executing a transaction will maintain logical consistency, temporal consistency can still be violated. For instance, take case (b) in the the example discussed earlier. Here, time has progressed to a point where *temperature* and *pressure* become temporally invalid even if they are logically consistent.

Thus, to satisfy *logical consistency* we use concurrency control techniques such as two phase locking [7] and to satisfy *temporal consistency* requirements we use *time-cognizant* transaction processing – by tailoring the traditional concurrency control and transaction management techniques to explicitly deal with time. To prepare the stage for discussing how this is done (in Section 5), we present the characteristics of transactions next.

3 Characteristics of Transactions in Real-Time Database Systems

In the first part of this section, transactions are characterized along three dimensions based on the nature of transactions in real-time database systems: the manner in which data is used by transactions, the nature of time constraints, and the significance of executing a transaction by its deadline, or more precisely, the consequence of missing specified time constraints. Subsequently, we show how the temporal consistency requirements of the data lead to some of the time constraints of transactions.

Real-time database systems employ all three types of transactions discussed in the database literature. For instance,

- *Write-only transactions* obtain state of the environment and write into the database.
- *Update transactions* derive new data and store in the database.
- *Read-only transactions* read data from the database and send them to actuators.

The above classification can be used to tailor the appropriate concurrency control schemes.

Some transaction time constraints come from temporal consistency requirements and some come from requirements imposed on system reaction time. The former typically take the form of periodicity requirements: For example,

Every 10 seconds Sample wind velocity.

Every 20 seconds Update robot position.

We show later in this section how the periodicity requirements can be derived from the *avi* of the data.

System reaction requirements typically take the form of deadline constraints imposed on aperiodic transactions: For example,

If $temperature > 1000$

within 10 seconds add coolant to reactor.

In this case, the system's action in response to the high temperature must be completed by 10 seconds.

Transactions can also be distinguished based on the effect of missing a transaction's deadline. In this paper, we use the terms *hard*, *soft* and *firm* to categorize the transactions. Viewed differently, this categorization tells us the *value* imparted to the system when a transaction meets its deadline. Whereas arbitrary types of value functions can be associated with activities [28], we confine ourselves to simple functions as described below.

- *Hard* deadline transactions are those which may result in a catastrophe if the deadline is missed. One can say that a large negative *value* is imparted to the system if a hard deadline is missed.

  These are typically safety-critical activities, such as those that respond to life or environment-threatening emergency situations.

- *Soft* deadline transactions have some value even after their deadlines. Typically, the value drops to zero at a certain point past the deadline. If this point is the same as the deadline, we get *firm* deadline transactions – which impart no value to the system once their deadlines expire [21].

  For example, if components of a transaction are assigned deadlines derived from the deadline of the transaction, then even if a component misses its deadline, the overall transaction might still be able to make its deadline. Hence these deadlines are soft. Another example is that of a transaction that is attempting to recognize a moving object. It must complete acquiring the necessary information before the object goes outside its view and hence has a firm deadline.

Figure 1 plots the value vs. time behavior of different types of transactions.

The processing of transactions must take their different characteristics into account. Since meeting time constraints is the goal, it is important to understand how transactions are scheduled and how their scheduling relates to time constraints. So in the rest of this section, we discuss how absolute validity requirements on the data induce periodicity requirements. As we shall see, it is not as straight-forward as it seems.

Suppose the *avi* of *temperature* is 10, i.e. *temperature* must be no more than 10 seconds old. Consider one of the many possible semantics of transactions with period P: One instance of the transaction must execute every period, as long as the start time and completion time lie within a period, the execution is considered to be correct with respect to the periodicity semantics. Suppose a simple transaction takes at most e units of time to complete, $(0 \leq e \leq P)$. Thus, if an instance starts at time t and ends at $(t + e)$ and the next instance starts at $(t + 2*P - e)$ and ends at $(t + 2*P)$, then we have two instances, which are separated by $(2*P)$ units of time in the worst case. This, for example, will be the case if the rate monotonic static priority approach, extended to deal with resources, [42, 43] is used to schedule periodic transactions executing on a main memory database. (Scheduling is discussed in greater detail in Section 5.) Thus, it follows from the above periodicity semantics that to maintain the *avi* of *temperature*, the period of the transaction that reads the *temperature* must be no more than half the *avi*, that is 5.

Let us assume instead that periodic transactions are scheduled so that each instance of a transaction is guaranteed to start at the same time, relative to the beginning of a period. Then, the worst case separation between the start time of one instance and the finish time of the subsequent instance will be $(P + (2*e))$. Since a transaction could write the relevant data item any time during its execution, the interval $(P + (2*e))$ must be less than the given *avi*. Thus, $P = (avi - (2*e))$.

The above discussion illustrates the dependence of transaction timing constraints not only on the temporal consistency requirements of the data but also on the execution times of the transactions and the scheduling approach adopted. The overall issue is one of *predictability* and we return to this in Section 5.

Now we consider deriving transactions' timing constraints from relative consistency specifications, recall that they must hold when a transaction uses the data in a relative consistency set to derive other data. So we must ensure that in an interval where such a transaction executes, from the point where relative consistency holds until the end of the interval, there is sufficient time for the transaction to complete execution. Handling *rvi*'s is clearly more involved [4]. Also, when we have a series of data derivations, each derivation being handled by a transaction, an alternative to using the *rvi*'s is to impose precedence constraints on the transactions to conform with the derived-from relationship. Much work remains to be done for methodically deriving transaction characteristics from the properties of the data.

4 Relationship to Active Databases

Many of the characteristics of data and transactions discussed in the last two sections may remind a reader of active databases. Hence this section is devoted to a discussion of the specific distinctions between active databases and real-time databases.

The basic building block in active databases is the following:

ON *event*
  IF *condition*
    DO *action*.

Upon the occurrence of the specified *event*, if the *condition* holds, then the specified *action* can be taken. This construct provides a good mechanism by which integrity constraints can be maintained among related or overlapping data or by which views can be constructed [13]. The *event* can be arbitrary, including external events (as in the case of real-time events generated by the environment), timer events, or transaction related events (such as the begin and commit of transactions). The *condition* can correspond to conditions on the state of the data or the environment. The *action* is said to be *triggered* [33, 12] and it can be an arbitrary transaction.

Given this, it is not difficult to see that active databases provide a good model for the *arrival* (i.e., triggering) of periodic/aperiodic activities based on events and conditions. Even though the above construct implies that an active database can be made to react to timeouts, time constraints are not *explicitly* considered by the underlying transaction processing mechanism.

However, as we have discussed before, the primary goal of real-time database systems is to *complete* the transactions on time. One can thus state the main deficiency in active databases in relation to what is required for them to deal with time constraints on the completion of transactions: time constraints must be *actively* taken into consideration.

Consider a system that controls the landing of an aircraft. Ideally, we would like to ensure that once the decision is made to prepare for landing, necessary steps, for example, to lower the wheels, to begin deceleration, and to reduce altitude, are completed within a given duration, say 10 seconds. Here the steps may depend on the landing path, the constraints specific to the airport, and the type of aircraft, and hence may involve access to a database containing the relevant information. In those situations where the necessary steps have not been completed in time, we would like to abort the landing within a given deadline, say within 5 seconds; the abort must be *completed* within the deadline, presumably because that is the "cushion" available to the system to take alternative actions. This requirement can be expressed as follows:

ON (*10 seconds after* "initiating landing preparations")
  IF (steps not completed)
    DO (*within 5 seconds* "Abort landing").

In summary, while active databases possess the necessary features to deal with many aspects of real-time database systems, the crucial missing ingredient is the active pursuit of the timely processing of actions.

5 Transaction Processing in Real-Time Database Systems

In this section, we discuss various aspects of transaction and query processing where the transactions and queries have characteristics discussed in Section 3, i.e, they have time constraints attached to them and there are different consequences of not satisfying those constraints.

A key issue in transaction processing is *predictability*. In the context of an individual transaction, this relates to the question: "will the transaction meet its time-constraint"? We discuss the sources of unpredictability in Section 5.1 and present ways by which the resulting problems can be addressed. Section 5.2 deals with the processing of transactions that have hard deadlines, while Section 5.3 deals with transactions that have soft deadlines.

5.1 The Need for Predictability

If a hard real-time transaction misses its deadline, it has catastrophic consequences. We can also say that missing the deadline has a large negative value to the system. Thus, we would like to *predict* beforehand that such transactions will complete before their deadlines. This prediction will be possible only if we know the worst-case execution time of a transaction and the data and resource needs of the transaction. In addition, it is desirable to have small variance between the worst-case predictions and the actual needs. Predictability is also important for soft deadline transactions, albeit to a lesser extent. In these cases, knowing before a transaction begins that the transaction may not complete within its deadline allows the system to discard the transaction, so that no time is spent on the transaction and no recovery overheads are incurred.

In a database system, a number of sources of unpredictability exist:

- Dependence of the transaction's execution sequence on data values;
- Data and resource conflicts;
- Dynamic paging and I/O; and
- Transaction aborts and the resulting rollbacks and restarts.

Distributed databases have additional problems due to communication delays and site failures. Below we elaborate upon these and point out ways by which individual problems can be alleviated. Finally, we outline a technique that is being developed to address these problems in the context of soft real-time transactions.

Since a transaction's execution path can depend on the values of the data items it accessed, it may not be possible to predict the worst-case execution time of the transaction. A similar problem arises for tasks in real-time systems. A similar solution applies: it is advisable to avoid use of unbounded loops and recursive or dynamically constructed data structures in real-time transactions. Since a real-time database is used in closed loop situations where the environment being controlled closes the loop, the data items accessed by a transaction are likely to be known once its functionality is known.

Since a typical transaction accesses data as it is needed in the execution sequence, it may be forced to wait until the data becomes available. Similarly, a transaction may be forced to wait for resources, such as CPU and I/O devices, to become available. While both these problems have their counterparts in real-time systems, the problems are exacerbated in real-time database systems due to data consistency requirements. Specifically, consider a database that employs strict two phase locking for concurrency control. In this case, a transaction may wait, in the worst case for an unbounded amount of time, when it attempts to acquire a data item. The cumulative delays can be very long; with deadlocks and restarts it could even be unbounded. Conflict avoiding data access protocols and the pre-allocation of resources have been developed to reduce this problem in real-time systems, but they do not apply directly to real-time database systems. We review some of these in Section 5.3 and show how it may be possible to adapt them in our context.

If disk-resident databases use demand-paged memory management, delays can occur while accessing disks both for fetching both data and program pages. These can lead to pessimistic worst-case scenarios since worst-case assumptions must be made about the need to fetch data or program page from disk whenever the need arises. This will depend on the disk scheduling and buffer management algorithms used. Main memory databases [1] eliminate these problems.

Transaction rollbacks also reduce predictability. Assume that a transaction is aborted and restarted a number of times before it commits. This has two negative consequences. The total execution time for the transaction increases and, if the number of aborts cannot be controlled, it may be unbounded. Second, the resources and time needed to handle the rollbacks will be denied to other transactions. Recovery time can be reduced by using semantics-based recovery discussed in Section 6. Real-time database systems may introduce transaction aborts due to deadline misses. One way to avoid these aborts is to begin a transaction only if we know that it will complete by its deadline. We give an overview of this approach below. Details can be found in [37].

Preanalysis of a transaction is desirable because it provides an estimate of its computation time and data and resource requirements. But, for complex transactions this may not be feasible. In this case, to get the necessary information about a transaction the following approach can prove useful. It has the potential to deal with the four sources of unpredictability mentioned above. Transactions go through two phases. In the first phase, called the pre-fetch phase, a transaction is run once, bringing in the necessary data into main memory if they are not in memory already. No writes are performed in this phase and conflicts with other transactions are not considered. The computational demands of the transactions are also determined during this phase. Assume that the data dependent portions of the transactions are such that a transaction's execution path does not change due to possible concurrent changes done to the data by other transactions while a transaction is going through its pre-fetch phase [15]. That is to say, at the end of the pre-fetch phase, all the necessary data is in memory. We now attempt to guarantee that the transaction will complete by its deadline. This is done by planning the execution of the transaction – respecting conflicts with the transactions already guaranteed – such that the transaction meets its deadline. This plan takes into account both the computational and resource requirements of the transaction and ensures that the necessary data and processing resources are available at the appropriate times for the transactions to complete within their time constraints. If such a plan cannot be constructed, the transaction is aborted without even starting it. The notion of guarantee and the planning algorithm are based on the resource constrained scheduling approach proposed for real-time systems and described in [39].

Let us see how this approach tackles the four major sources of unpredictability mentioned above. By using the pre-fetch phase to bring in the pages, the actual execution sequence is determined during this phase. Data and resource conflicts during execution are avoided by the use of explicit planning of the execution phase of transactions. Since necessary pages are brought into memory during the pre-fetch phase, dynamic I/O is avoided during the execution phase. Finally, transaction aborts and rollbacks are avoided because all changes are done during the execution phase and this phase is not begun unless it is known that it will complete in time.

If the state of the data changes during the pre-fetch phase, which can be detected by detecting the writes to the data brought into memory by the transaction, then the pre-fetch phase can be reexecuted. In any case, this approach provides a way by which if access invariance holds, once guaranteed, a transaction will complete by its deadline and no recovery actions are necessary if a transaction is unable to execute. The price paid in the latter situation is the overheads of the pre-fetch phase. Several optimizations are possible. For example, in some situations, there may not even be a need to go to the execution phase. As in optimistic concurrency control this will happen if the data items used by the transaction were not used by any other concurrent transaction. Details can be found in [37].

5.2 Dealing with Hard Deadlines

All transactions with hard deadlines must meet their time constraints. Since dynamically managed transactions cannot provide such a guarantee, the data and processing resources as well as time needed by such transactions have to be guaranteed to be made available when necessary. There are several implications of this.

Firstly, we have to know when the transactions are likely to be invoked. This information is readily available for periodic transactions, but for aperiodic transactions, by definition, it is not. The smallest separation time between two incarnations of an aperiodic transaction can be viewed as its period. Thus, we can cast all hard real-time transactions as *periodic* transactions.

Secondly, in order to ensure a priori that their deadlines will be met, we have to determine their resource requirements and worst-case transaction execution times. As outlined in Section 5.1, this requires that many restrictions be placed on the structure and characteristics of real-time transactions.

Once we have achieved the above, we can treat the transactions in a manner similar to the way real-time systems treat periodic tasks that require guarantees, i.e., by using static *table-driven* schedulers or preemptive *priority-driven* approaches. Static *table-driven* schedulers reserve specific time slots for each transaction. If a transaction does not use all of the time reserved for it, the time may be reclaimed [44] to start other hard real-time transactions earlier than planned. Otherwise, it can be used for soft real-time transactions or left idle. The table-driven approach is obviously very inflexible. A priority-driven approach is the rate-monotonic priority assignment policy. One can apply the schedulability analysis tools associated with it to check if a set of transactions are schedulable given their periods and data requirements. This is the approach discussed in [43] where periodic transactions that access main memory resident data via read and write locks are scheduled using rate-monotonic priority assignment.

We mentioned earlier that the variance between the worst-case computational needs and actual needs must not be very large. We can see why. Since the schedulability analysis is done with respect to worst-case needs, if the variance is large, many transactions that may be doable in the average case will be considered infeasible in the worst-case. Also, if the table-driven approach is used, a large variance will lead to large idle times.

In summary, while it is possible to deal with hard real-time transactions using approaches similar to those used in real-time systems, many restrictions have to be placed on these transactions so that their characteristics are known a priori. Even if one is willing to deal with these restrictions, poor resource utilization may result given the worst-case assumptions made about the activities.

5.3 Dealing with Soft Deadlines

With soft real-time transactions, we have more leeway to process transactions since we are not required to meet the deadlines all the time. Of course, the larger the number of transactions that meet their deadlines the better. When transactions have different values, the value of transactions that finish by their deadlines should be maximized. The complexity involved in processing real-time transactions comes from these goals. That is to say, we cannot simply let a transaction run, as we would in a traditional database system, and abort it should its deadline expire before it commits. As we discussed in Section 4, we must *actively* pursue the goal of meeting transaction deadlines by adopting priority-assignment policies and conflict resolution mechanisms that explicitly take time into account. Note that priority assignment governs CPU scheduling and conflict resolution determines which of the many transactions contending for a data item will obtain access. As we will see, conflict resolution protocols make use of transaction priorities and because of this, the priority assignment policy plays a crucial role [25]. We discuss these two issues in Section 5.3.1. We also discuss the performance implications of different deadline semantics. Additional aspects of transaction management, such as, distribution, transaction commitment, and deadlock detection are discussed in Section 5.3.2.

5.3.1 Priority Assignment and Conflict Resolution

Rather than assigning priorities based on whether the transactions are CPU or I/O (or data) bound, real-time database systems must assign priorities based on transaction time constraints and the value they impart to the system. Possible policies are:

- *Earliest-deadline-first.*
- *Highest-value-first.*
- *Highest-value-per-unit-computation-time-first.*
- *Longest-executed-transaction-first*

It has been shown that the priority assignment policy has significant impact on performance and that when different transactions have different values, both deadline *and* value must be considered [25].

For the purpose of conflict resolution in real-time database systems, various *time-cognizant* extensions of two phase locking, optimistic, and timestamp based protocols have been proposed in the literature [1, 2, 9, 20, 25, 27, 26, 34, 47, 49]. These are discussed below.

In the context of two-phase locking, when a transaction requests a lock that is currently held by another transaction we must take into account the characteristics of the transactions involved in the conflict. Considerations involved in conflict resolution are the deadline and value (in general, the priority) of transactions, how long the transactions have executed, and how close they are to completion. Consider the following set of protocols investigated in [27].

- If a transaction with a higher priority is forced to wait for a lower priority transaction to release the lock, a situation known as *priority inversion* arises. This is because a lower priority transaction makes a higher priority transaction to wait. In one approach to resolving this problem, the lock holder *inherits* the lock requester's priority whereby it completes execution sooner than with its own priority.

- If the lock holding transaction has lower priority, abort it. Otherwise let the lock requester wait.

- If the lock holding transaction is closer to its deadline, lock requester waits, independent of its priority.

Priority Inheritance is shown to reduce transaction blocking times [27]. This is because the lock holder executes at a higher priority (than that of the waiting transaction) and hence finishes early, thereby blocking the waiting higher priority transaction for a shorter duration. However, even with this policy, the higher priority transaction is blocked, in the worst case, for the duration of a transaction under strict two phase locking. As a result, the priority inheritance protocol typically performs even worse than a protocol that makes a lock requester wait independent of its priority.

If a higher priority transaction always aborts a low priority transaction, the resulting performance is sensitive to data contention. On the other hand, if a lower priority transaction that is closer to completion inherits priority rather than aborting, then a better performance results even when data contention is high. Such a protocol is a combination of the abort-based protocol proposed for traditional databases [50] and the priority-inheritance protocol proposed for real-time systems [42]. Said differently, the superior performance of this protocol [27] shows that even though techniques that work in real-time systems on the one hand and database systems on the other hand may not be applicable directly, they can often be tailored and adapted to suit the needs of real-time database systems. It should be noted that abort-based protocols (as opposed to wait-based) are especially appropriate for real-time database systems because of the time constraints associated with transactions.

Let us now consider optimistic protocols. In protocols that perform backward validation, the validating transaction either commits or aborts depending on whether it has conflicts with transactions that have already committed. The disadvantage of backward validation is that it does not allow us to take transaction characteristics into account. This disadvantage does not apply to forward validation. In forward validation, a committing transaction usually aborts ongoing transactions in case they conflict with the validating transaction. However, depending on the characteristics of the validating transaction and those with which it conflicts, we may prefer not to commit the validating transaction. Several policies have been studied in the literature [20, 21, 26]. In one, termed *wait-50*, a validating transaction is made to wait as long as more than half the transactions that conflict with it have earlier deadlines. This is shown to have superior performance.

Time-cognizant extensions to timestamp-based protocols have also been proposed. In these, when data accesses are out of timestamp order, the conflicts are resolved based on their priorities. In addition, several combinations of locking-based, optimistic and timestamp-based protocols have been proposed but require quantitative evaluation [34].

Exploiting multiple versions of data for enhanced performance has been addressed in [29]. Multiple versions can reduce conflicts over data. However, if data must have temporal validity, old versions which are outdated must be discarded. Also, when choosing versions of related data, their relative consistency requirements must be taken into account: consider a transaction that uses multi-versioned data to display aircraft positions on an air-traffic controller's screen. The data displayed must have both absolute validity as well as relative validity.

Different transaction semantics are possible with respect to discarding a transaction once its deadline is past. For example, with firm deadlines, a late transaction is aborted once its deadline expires [21]. In general, with soft deadlines, once a transaction's value drops to zero, it is aborted [25]. On the other hand, in the transaction model assumed in [1], all transactions have to complete execution even if their deadlines have expired. In this model, delayed transactions may cause other transactions also to miss their deadlines and this can have a cascading effect. Needless to say, it is important to exploit transaction semantics so as to abort them as soon as it is clear that there is little benefit to continuing the execution of a transaction. Of course, aborting a transaction also has performance implications given the costs of recovery. We discuss this in Section 6.3.

Before we end this section, it should be pointed out that special time-cognizant deadlock detection, transaction wakeup, and restart policies appear to have little impact [25]. For example, breaking a deadlock cycle by aborting a transaction based on transaction timing characteristics does not seem to produce significantly better results. Similarly, which of many rolled back transactions to restart next or which of many waiting transactions to wakeup next can be determined by taking transaction's timing characteristics into account. However, in many situations tested to date, the differences between the possible choices do not seem to warrant special handling of restarts or wakeups.

5.3.2 Commitment, Distribution, and Nested Transactions

Let us now consider the transaction commitment process. Once a transaction reaches its commit point, it is better to let it commit quickly so that its locks can be released soon. If commit delays are not high, which will be the case in a centralized database, the committing transaction can be given a high enough priority so that it can complete quickly. The solution is not so easy in a distributed system because of the distribution of the commitment process. Furthermore, since a deadline typically refers to the deadline until the end of the two-phase commit, but since the decision on whether or not to commit is taken in the first phase, we can enter the second phase only if we know that it will complete before the deadline. This requires special handling of the commit process. An alternative is to associate the deadline with the beginning of the second phase, but this may delay subsequent transactions since locks are not released until the second phase.

A distributed real-time database system introduces other complications as well, especially when we go beyond flat transactions. Let us consider nested transactions [36]. Even though transaction models that are more complex than flat transactions introduce additional unpredictability, some activities with soft time constraints may find them more suitable since, for instance, nested transactions allow the independent recovery of subtransactions.

So far we assumed that each transaction has a value and a deadline. These can be used in several ways in the nested transaction model.

- Suppose we assign a deadline and value only to the top-level transaction. Some scheme will have to be designed to propagate these to the nested child transactions, to their children, and so on, so that conflicts between the components of a nested transaction and other transactions can be dealt with as though they were separate transactions.

  Knowledge of computation times of (child) transactions will prove useful in appropriately assigning the intermediate deadlines of the child transactions. The deadline for a child transaction should depend on the deadline of the top-level transaction, the computation time of the transaction and its children, as well as the system load.

- Suppose individual deadlines and values are assigned to each component of a nested transaction. Then the system will have to "reassign" the value and the deadline so that they are consistent with each other, for example, to make sure that the deadline of a parent is no earlier than that of its children.

The former is more applicable to multi-level transactions where nesting is implicit and is hidden from the user and the latter more applicable to nested transactions where the nesting structure is visible to the user. In either case, deadlines associated with children have implications when a deadline is missed. Since it is the top-level transaction that must meet its deadline, it may be possible for children to miss deadlines and yet the top-level transaction may meet its deadline. That is, the deadlines for the children are soft deadlines. In certain situations, it may be possible to abort a delayed child and run an alternative child transaction instead.

In a flat transaction model, transactions are competing against each other for data as well as computational and I/O resources, but components of a nested transaction, even if they have individual deadlines, are executing on behalf of that transaction. Hence scheduling and conflict resolution strategies have to be tailored to handle the case of components of the same nested transaction competing with each other. Further problems arise when components of a nested transaction execute on different sites. Specifically, transaction priorities must be set in a *consistent* fashion at all the sites visited by a transaction (or its components).

A related topic is the replication of data. Its potential for fault-tolerance is an especially important one for distributed real-time database systems. However, very little work has been done to-date on this and other issues raised above for distributed real-time databases or for transaction models beyond flat transactions.

6 Other Issues in Real-Time Database Systems

In this section, we would like to bring together a number of issues that have not been adequately addressed in the real-time database literature. These include managing resources other than CPU and data, trading off timeliness for quality, managing recovery, and handling overloads. The subsections in this section deal with these topics individually. Since little work has been done in these areas, the discussion is, by necessity, speculative.

6.1 Managing I/O and Buffers

Whereas the scheduling of CPU and data resources has been studied fairly extensively in the real-time database literature, studies of scheduling approaches for dealing with other resources, such as disk I/O, and buffers has begun only recently. In this section we review some recent work in this area and discuss some of the problems that remain.

I/O scheduling is an important area for real-time systems given the large difference in speeds between CPU and disks and the resultant impact of I/O devices' responsiveness on performance. However, real-time systems research has essentially ignored this problem because of the perception that disk access introduces high degree of unpredictability and so disks are seldom accessed when time constraints exist. However, in real-time database systems the reading and writing of (archival) data is essential and so disk scheduling when transactions have time constraints becomes a significant problem. Since the traditional disk scheduling algorithms attempt to minimize average I/O delays, just like traditional CPU scheduling algorithms aim to minimize average processing delays, time-cognizant I/O scheduling approaches are needed.

It must be recognized that what is important is the meeting of transaction deadlines and not the individual deadlines that may be attached to I/O requests. Assume that we model a transaction execution as a sequence of (disk I/O, computation) pairs culminating in a set of disk I/O's, the latter arising from writes to log and to the changed pages. Suppose we assign (intermediate) deadlines to the I/O requests of a transaction given the transaction's deadline. One of the interesting questions with regard to disk I/O scheduling is: How does one derive the deadline for an I/O request from that of the requesting transaction? First of all, it must be recognized that depending on how these I/O deadlines are set, deadlines associated with I/O requests may be *soft* since even if a particular I/O deadline is missed, the transaction may still complete by its deadline. This is the case if I/O deadlines are set such that the overall laxity (i.e., the difference between the time available before the deadline and the total computation time) of a transaction is uniformly divided among the computations and the I/O. On the other hand, assume that an intermediate deadline is equal to the latest completion time (i.e., the time an I/O must complete assuming that subsequent computations and I/O are executed without delay). This is the less preferred method since we now have a firm deadline associated with I/O requests – if an I/O deadline is missed, there is no way for the transaction to complete by its deadline and so the requesting transaction must be aborted.

Recent work on I/O scheduling includes [10, 3, 11]. The priority driven algorithm described in [10] is a variant of the traditional SCAN algorithm which works on the elevator principle to minimize disk arm movement. Without specifying how priorities are assigned to individual I/O requests, [10] proposes a variant in which the SCAN algorithm is applied to each priority level. Requests at lower priority are serviced only after those at higher priority are served. Thus, if after servicing a request, one or more higher priority requests are found waiting, the disk arm moves towards the highest priority request that is closest to the current disk arm position. In the case of requests arising from transactions with deadlines, priority assignment could be based on the deadline assigned to the I/O request.

Another variant of SCAN, one which directly takes I/O deadlines into account is FD-

SCAN [3]. In this algorithm, given the current position of the disk arm, the disk arm moves towards the request with the earliest deadline that can be serviced in time. Requests that lie in that direction are serviced and after each service it is checked whether (1) a request with an even earlier deadline has arrived and (2) the deadline of the original result cannot be met. In either case, the direction of disk arm movement may change.

Clearly, both these protocols involve checks after each request is served and so incur substantial run-time overheads. The protocols described in [11] are aimed at avoiding the impact of these checks on I/O performance. Specifically, the protocols perform the necessary computations while I/O is being performed. In the SSEDO algorithm (Shortest-seek and Earliest Deadline by Ordering), the need to give higher priority to requests with earlier deadlines is met while reducing the overall seek times. The latter is accomplished by giving a high priority to requests which may have large deadlines but are very close to the current position of the disk arm. A variant of SSEDO is SSEDV which works with specific Deadline Values, rather than Deadline Orderings. [11] shows how both the algorithms can be implemented so as to perform disk scheduling while service is in progress and shows that the algorithms have better performance than the other variants of the SCAN algorithms.

Another resource for which contention can arise is the database buffer. What we have here is a conflict over buffer slots – akin to conflicts that occur over a time slot, in the case of a CPU. Thus, similar issues arise here also. Specifically, how to allocate buffer slots to transactions and which slots to replace when a need arises are some of the issues. Consider buffer replacement: in case there is a need to replace an existing buffer slot to make room for a new entry, the replacement policy may have an impact on performance, especially if the slot being replaced is used by an uncommitted transaction. Work done in this area includes [24, 10]. Whereas [24] reports of no significant performance improvements when time-cognizant buffer management policies are used, studies discussed in [10] show that transaction priorities must be considered in buffer management. Clearly, the jury is still out on the issue and further work is needed.

6.2 Performance Enhancement: Trading off Quality for Timeliness

Before we examine the specific performance enhancement possibilities unique to real-time database systems, it is important to point out that several proposals made for performance enhancement in traditional databases are also applicable to real-time databases. For instance, given that the data objects in real-time database systems will be abstract data type objects, as opposed to read/write objects, the semantics of the operations on these objects can be exploited to improve concurrent access to these objects (see, for example, [5]). Generalizing this, the parallelism and distribution inherent in real-time systems, which by their very nature function in physically distributed environments with multiple active processing elements, can be put to use to improve performance. Of course, as we discussed earlier, distribution brings with it some special problems in the real-time context. With regard to predictability many advantages can be gained by the use of main memory databases. Also, the benefits afforded by database machines [30] for real-time database systems are worth exploring.

Now let us consider approaches that are in some sense unique to real-time database systems. In the context of activities having timing constraints, the statement, "it is better to produce a partial result before the deadline instead of the complete result after the deadline" has become a cliche. However, it is not always clear what an acceptable partial result is or how a computation can be structured to provide acceptable partial results. Recent work in the real-time area can lead us to some partial answers [35]. In general, *timeliness*, a key performance measure, could be achieved by trading it off with completeness, accuracy, consistency, and currency [19, 40]. Below we consider each of these in turn.

Let us first consider *completeness*. Suppose a transaction updates the screen of an operator in a chemical plant periodically. If during a certain time interval, during overloads, it is unable to update all the valve positions, but has the time to update those that are crucial to the safety of the plant, then such a transaction should be allowed to execute even if not all its actions may be performed.

When query processing involves computing aggregates, especially in a time-constrained environment, then one can achieve different degrees of *accuracy* by resorting to approximate query processing by sampling data [23]. Here, depending on time availability, results with different accuracies can be provided. Another example is that of a transaction that does not have all the necessary data for its processing but can recover from this situation by extrapolating based on previous data values. Here again, if previous data values of different data items are used, their relatively consistency must be considered.

Turning to *consistency*, in the context of traditional databases, it has often been mentioned that correctness notions that relax serializability are appropriate (see [41] for a review of such relaxed notions.). For instance, epsilon serializability [38] allows a query to execute in spite of concurrent updates wherein the deviation of the query's results, from that of a serializable result, can be bounded. Such relaxations allow more transactions to execute concurrently thereby improving performance.

In the context of *currency* of a transaction's results it may not always be necessary for a transaction to use the latest version of a data item. This is true, for example, when a transaction is attempting to derive trends in the changes to some data. Clearly, old versions of the data are required here and the transaction can complete even if the latest version is unavailable.

The examples mentioned above make it clear that there are situations where imprecision can be tolerated, and in fact must be exploited, to improve performance. However, how to achieve this systematically is yet to be studied. What we need are notions similar to the degrees of consistency adopted in traditional database systems [17]. In this context, scheduling approaches that have been developed for the imprecise computation model in real-time systems could be tailored to apply to real-time database systems. Preliminary work in this area is reported in [45].

6.3 Recovery Issues

Recovery is a complex issue even in traditional databases and is more so in real-time database systems for two reasons. (The approach discussed at the end of Section 5.1 was motivated in part by these complexities.) Firstly, the process of recovery can interfere with the processing of ongoing transactions. Specifically, suppose we are recovering from a transaction aborted due to a deadline miss. If locks are used for concurrency control, it is important to release them as soon as possible so that waiting transactions can proceed without delay so as to meet their deadlines. However, it is also necessary to undo the changes done by the transaction to the data if in-place updates are done. But this consumes processing time that can affect the processing of transactions that are not waiting for locks to be released. Whereas optimistic concurrency control techniques or a shadow-pages based recovery strategy can be used to minimize this time, they have several disadvantages [18]. Secondly, unlike traditional databases where permanent data should always reflect a consistent state, in real-time databases, the presence of temporal data, while providing some opportunities for quicker recovery [51], adds to the complexities of the recovery of transactions. Specifically, if a transaction's deadline expires before it completes the derivation of a data item, then rather than restoring the state of the data to its previous value, it could declare the data to be invalid thereby disallowing other transactions from using the value. The next instance of the transaction, in case the data is updated by a periodic transaction, may produce a valid state.

In general, real-time database recovery must consider time and resource availability to determine the most opportune time to do recovery without jeopardizing ongoing transactions, whether they are waiting for locks or not. Available transaction as well as data semantics (or state) must be exploited to minimize recovery overheads. Contingency or compensating transactions [32] are applicable here: Contingency transactions can take the form of multiple versions of a transaction each with different values and different computational and data requirements. If we know that one with the highest quality will be unable to complete in time, the system can recover by trying an alternative with acceptable quality. This is a situation where quality is traded off to minimize recovery costs and to achieve timeliness. Revisiting the factory floor example from the introduction, we saw that if there is insufficient time to complete object recognition, the system discards the object for now and directs the object to appear once again in front of the camera (at perhaps a later point in time). In case a real-time transaction has interacted with the environment, a compensating transaction may have to be invoked to recover from its failure [32]. The nature and state of the environment can be used to determine recovery strategies. In some situations, in the absence of new data that was to have been produced by an aborted transaction, extrapolation of new values from old values may be possible. In other cases, more up-to-date data may be available soon.

The following highly simplified example may help in illustrating some of the considerations in recovery. Suppose two robots on a factory floor have to rendezvous at point x by time t: t is a firm deadline by which either both should be at x or both should know that they cannot make it. The controller of the robot, i.e., the real-time system, first obtains their current position and those of the pertinent objects on the factory floor. It determines the type of moves the robots are capable of by retrieving their characteristics from archival storage. It then creates a path for each robot to follow to reach x by time t and sends this path to each robot. The controller also reserves this path for the duration for these two robots. As the robots follow this path, the controller monitors their movement, looks out for obstacles in their slated path and continually checks if there is a delay in reaching specific points along the path due to incorrect estimations made during path construction or unanticipated other delays. If it detects such a situation, the controller recovers from it by determining an alternative path given the robots' current position. Should there be no time to follow the new path, recovery involves instructing the robots to halt, informing each of them that their rendezvous is not possible. In either case, path reservation information is modified appropriately. Note that all of this involves reading information from the environment, retrieving information from the database, and updating other information. It also shows some aspects of recovery: Recovery here comprises two contingency actions, one of which involves termination of the transaction after informing the robots.

6.4 Managing Overloads

Perhaps the most critical of the outstanding issues is one of managing overloads. How should real-time transaction processing be done when more transactions arrive than can meet their deadlines? In traditional systems, if an overload does not remain for too long, in most cases, the result is a slow response for the duration of the overload. However, in real-time databases that interact with the environment, catastrophic consequences can arise. These can be minimized by ensuring that transactions that are critical to the performance of the system are declared to possess hard deadlines and are guaranteed to meet deadlines even under overloads. In addition, if we make sure that transaction values are considered for priority assignment and during conflict resolution, then the transaction that misses its deadline will typically have a low value. However, missing too many low-valued transactions with soft deadlines may eventually lead to situations where many transactions with high values arrive thus stressing the system: For example, if periodic maintenance is postponed due to the arrival of more important activities, it may eventually be necessary to shut down the system. Hence dealing with overleads is complex and solutions are still in their infancy [6, 8, 31]. An approach to this problem, based on discarding transactions immediately upon their arrival, given current system load and arriving transaction characteristics, is described in [22]. In managing overloads, some of the tradeoffs that we discussed earlier, involving timeliness vs. quality are also very pertinent.

7 Conclusions

In this paper, we presented the characteristics of data and transactions in real-time database systems and discussed the differences between real-time database systems and traditional databases. Many of the differences arise because temporal consistency requirements are imposed on the data in addition to the usual integrity constraints. Maintaining temporal consistency imposes time constraints on the database transactions. In addition, the reaction requirements demanded by the environment can also place time constraints. The performance of real-time database systems is measured by how well the time constraints associated with transactions are met. The system must meet all hard deadlines and minimize the number of transactions whose soft deadlines are missed. This is a crucial difference from traditional databases and necessitates *time-cognizant* transaction processing.

We examined various aspects of transaction processing in real-time database systems including concurrency control and recovery and showed that recovery becomes an even more complex problem when transactions have time constraints. In many situations, one can trade off timeliness for quality of the transaction's results where the quality depends on the completeness, accuracy, currency, and consistency of the results. Furthermore, many recent advances in databases, for exploiting parallelism, distribution, object semantics, and transaction semantics should be very useful in real-time database systems also.

Whereas recently there has been a spurt of research activity in the area, many open questions remain. These include the derivation of transaction timing properties from the temporal consistency requirements of the data, developing suitable hardware and software architectures for real-time database systems, seamless management of transactions with hard and soft deadlines, real-time transaction processing in distributed databases, transaction recovery, and the tradeoffs between timeliness and quality.

Acknowledgements

I am indebted to real-time database researchers for many of the ideas discussed in this paper. Particular thanks to A. Buchmann, K. Dittrich, C. Mohan, and S. Son who participated in the panel on real-time database systems at VLDB in Barcelona. Thanks also to Azer Bestavros, Panos Chrysanthis, Jayant Haritsa, Lory Molesky, Patrick O'Neil, Bhaskar Purimetla, Dennis Shasha, Chia Shen, Chia-Shiang Shih, Jack Stankovic, Zhao Wei, and the anonymous reviewers, for their comments on previous versions of this paper.

References

[1] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proceedings of the 14th VLDB Conference*, Aug. 1988.

[2] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *Proceedings of the 15th VLDB Conference*, 1989.

[3] R. Abbott and H. Garcia-Molina, "Scheduling I/O Requests with Deadlines: A Performance Evaluation," *Proceedings of the Real-Time Systems Symposium*, Dec. 1990.

[4] N. Audsley, A. Burns, M. Richardson, and A. Wellings, "A Database Model for Hard Real-Time Systems", Technical Report, Real-Time Systems Group, Univ. of York, U.K., July 1991.

[5] B. R. Badrinath and K. Ramamritham. "Semantics-Based Concurrency Control: Beyond Commutativity," *ACM Transactions on Database Systems*, March 1992.

[6] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, F. Wang, "On the Competitiveness of On-Line Real-Time Scheduling", *Proceedings of the Real-Time Systems Symposium*, December 1991.

[7] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[8] S. Biyabani, J.A. Stankovic, and K. Ramamritham, "The Integration of Deadline and Criticalness in Hard Real–Time Scheduling," *Proceedings of the Real-Time Systems Symposium*, December 1988.

[9] A.P. Buchmann, D.R. McCarthy, M. Chu, and U. Dayal, "Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control", *Proceedings of the Conference on Data Engineering*, 1989.

[10] M.J. Carey, R. Jauhari, and M. Livny, "Priority in DBMS Resource Scheduling", *Proceedings of the 15th VLDB Conference*, Aug 1989, pp. 397-410.

[11] S. Chen, J. Stankovic, J. Kurose, and D. Towsley, "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems", *Real-Time Systems*, Sept. 1991.

[12] U. Dayal, et. al. "The HiPAC Project: Combining Active Databases and Timing Constraints", *SIGMOD Record*, 17, 1, March 1988, 51-70.

[13] K. R. Dittrich and U. Dayal. Active Database Systems (Tutorial Notes). In *The Seventeenth International Conference on Very Large Databases*, September 1991.

[14] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notion of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[15] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian, "Access Invariance and its Use in High Contention Environments", *Proceedings of the Sixth International Conference on Database Engineering*, 1990, pp 47-55.

[16] M.C. Graham. "Issues in Real-Time Data Management", CMU/SEI-91-TR-17, July 1991.

[17] J. N. Gray, R. A. Lorie, G. R. Putzulo, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared database. In *Proceedings of the First International Conference on Very Large Databases*, pages 25–33, Framingham, MA, September 1975.

[18] J.N. Gray and A. Reuter, "Transaction Processing: Techniques and Concepts", Morgan-Kaufman (book in preparation).

[19] N. Griffeth and A. Weinrib, "Scalability of a Real-Time Distributed Resource Counter", Proceedings of the Real-Time Systems Symposium, Orlando, Florida (December 1990).

[20] J.R. Haritsa, M.J. Carey and M. Livny, "On Being Optimistic about Real-Time Constraints," *Proceedings of ACM PODS*, 1990.

[21] J.R. Haritsa, M.J. Carey and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *Proceedings of the Real-Time Systems Symposium*, Dec. 1990.

[22] J.R. Haritsa, M.J. Carey and M. Livny, "Earliest Deadline Scheduling for Real-Time Database Systems," *Proceedings of the Real-Time Systems Symposium*, Dec. 1991.

[23] W. Hou, G. Ozsoyoglu, B. K. Taneja, "Processing Aggregate Relational Queries with Hard Time Constraints", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, June 1989.

[24] J. Huang and J. Stankovic, "Real-Time Buffer Management," COINS TR 90-65, August 1990.

[25] J. Huang, J.A. Stankovic, D. Towsley and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," *Proceedings of the Real-Time Systems Symposium*, Dec. 1989

[26] J. Huang, J.A. Stankovic, K. Ramamritham and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes", *Proceedings of the Conference on Very Large Data Bases*, Sep 1991.

[27] J. Huang, J.A. Stankovic, K. Ramamritham and D. Towsley, "On Using Priority Inheritance in Real-Time Databases," *Proceedings of the Real-Time Systems Symposium* December 1991.

[28] Jensen, E. D., Locke, C. D. and Tokuda, H., "A Time-Driven Scheduling Model For Real-Time Operating Systems", *Proceedings of 1985 IEEE Real-Time Systems Symposium,* pp. 112-122.

[29] W. Kim and J. Srivastava, "Enhancing Real-Time DBMS Performance with Multiversion Data and Priority-Based Disk Scheduling", *Proceedings of the Real-Time Systems Symposium*, Dec 1991, pp. 222-231.

[30] Kitsurgawa, "The next generation of database machines", this issue.

[31] G. Koren and D. Shasha, "D-Over: an optimal on-line scheduling algorithm for overloaded real-time systems" *Real-Time Systems Symposium*, Dec 1992.

[32] H. F. Korth, E. Levy, and A. Silberschatz. Compensating Transactions: A New Recovery Paradigm. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 95–106, Brisbane, Australia, August 1990.

[33] H. F. Korth, Soparkar, Silberschatz, A. "Triggered Real-Time databases with consistency constraints", *Proceedings of the Conference on Very Large Data Bases*, 1990.

[34] Y. Lin and S.H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *Proceedings of the Real-Time Systems Symposium*, Dec. 1990.

[35] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zhao, "Algorithms for Scheduling Imprecise Computation", *IEEE Computer*, Vol. 24, No. 5, May 1991.

[36] J. E. B. Moss, *Nested Transactions: An approach to reliable distributed computing.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, April 1981.

[37] P. O'Neil, K. Ramamritham, and C. Pu, "Towards Predictable Transaction Executions in Real-Time Database Systems", Technical Report 92-35, University of Massachusetts, August, 1992.

[38] C. Pu and A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 377–386, May 1991.

[39] K. Ramamritham, J. Stankovic, and P. Shiah, "Efficient Scheduling Algorithms for Real–Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, April 1990, pp. 184-194.

[40] K. Ramamritham, S. Son, A. Buchmann, K. Dittrich, and C. Mohan, "Real-Time Databases" panel statement, *Proceedings of the Conference on Very-Large Databases*, September, 1991.

[41] K. Ramamritham and P. Chrysanthis, "In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties" in *Distributed Object Management*, Ozsu, Dayal, and Valduriez Ed., Morgan Kaufmann Publishers, 1992.

[42] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *IEEE Transactions on Computers*, 39(9), pp. 1175-1185, 1990.

[43] L. Sha, R. Rajkumar and J.P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record*, March 1988.

[44] C. Shen, K. Ramamritham, and J. Stankovic, Resource Reclaiming in Real-Time, *Proc Real-Time System Symposium*, December 1990. (to appear in *IEEE Transactions on Parallel and Distributed Systems*).

[45] K. P. Smith and J.W.S. Liu, "Monotonically improving approximate answers to relational algebra queries", *Proceedings of Compsac*, September 1989.

[46] R. Snodgrass and I. Ahn, "Temporal Databases", *IEEE Computer*, Vol 19, No. 9, September 1986, pp. 35-42.

[47] S. .H. Son, Y. Lin, and R. P. Cook, "Concurrency Control in Real-Time Database Systems", in *Foundations of Real-Time Computing: Scheduling and Resource Management*, edited by Andre van Tilborg and Gary Koob, Kluwer Academic Publishers, pp. 185-202, 1991.

[48] X. Song and J.W.S. Liu, "How Well Can Data Temporal Consistency be Maintained?" *Proceedings of the IEEE Symposium on Computer-Aided Control Systems Design*, (to appear) 1992.

[49] J.A. Stankovic, K. Ramamritham, and D. Towsley, "Scheduling in Real-Time Transaction Systems," in *Foundations of Real-Time Computing: Scheduling and Resource Management*, edited by Andre van Tilborg and Gary Koob, Kluwer Academic Publishers, pp. 157-184, 1991.

[50] Y. C. Tay and Nathan Goodman and Rajan Suri, "Locking performance in centralized databases", *ACM Transactions on Database Systems*, volume 10, number 4, December, 1985, pp. 415–462.

[51] S. V. Vrbsky and K.J. Lin. "Recovering Imprecise Computations with Real-Time Constraints", *Proceedings of the Seventh Symp. on Reliable Distributed Systems*, October 1988, pp. 185-193.