Technical Report

CMU/SEI-91-TR-017
ESD-91-TR-017

# Issues In Real-Time Data Management

Marc H. Graham

July 1991

**Technical Report**

**CMU/SEI-91-TR-017**<br>
**ESD-91-TR-017**<br>
**July 1991**

# Issues in Real-Time Data Management

**Marc H. Graham**

Rate Monotonic Analysis for Real-Time Systems Project

**Software Engineering Institute**
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

# Issues in Real-Time Data Management

**Abstract:** This report explores issues related to the use of database management technology in support of real-time system programming. It describes the potential benefits of database support for real-time systems, and it describes the state of the art in database technologies relevant to real-time. The report concludes that more research and development will be needed before the benefits of database management can be applied to real-time system development.

# 1. Introduction

This paper explores real-time data and database management with the goal of determining whether and how the engineering of real-time systems can be improved through exploitation of these technologies. Improving the engineering of real-time systems means decreasing the cost of construction and maintenance and increasing the reliability of the product. A powerful technique to accomplish that improvement is to substitute off-the-shelf technology for customized technology.

Database management systems (DBMSs) handle shared, persistent data; real-time systems are more usually concerned with transient data with very short meaningful lifetimes. However, *persistent* and *transient*, *shared* and *exclusive* are relative terms. Data may be considered shared and persistent if it lasts long enough to be used by subsystems written by different people or organizations. Database technology facilitates communication among engineers, not just among their products.

Stankovic maintains [52] that real-time computing is not high-performance computing, but rather computing with time constraints. The correctness criteria of a real-time system include explicit statements about the time in which the computing must be done.[1] In order to engineer a system that meets its time constraints, it must be possible to analyze the system to determine its time consumption. Commercial, off-the-shelf (COTS) database technology cannot be analyzed in this way because the producers of that technology will publish neither the analysis (if they have it) nor the code, in order to protect it. There appears to be no example of a successfully fielded, real-time database application using any commercial, off-the-shelf DBMS. This may reflect the fact that practitioners are too busy to publish their practices. However, some successful industrial R&D efforts that "would have worked had they been fielded" have been noted [26, 44]. A British software house that produces systems for the British Navy has developed software for real-time database management [58], but the British consider it a competitive advantage and are unwilling to discuss it.

[1] For every system, there is a time constraint that, if not met, makes the system unusable. Consider a word processing system with a thirty second response to a keystroke. In that sense, every system is a real-time system. We say a system is a real-time system if the time constraints are explicit and quantitative.

The current state of the practice in real-time data management relies heavily on customization and innovation and little on routine, off-the-shelf technology. This situation cannot be reversed in the near term. As I will demonstrate, not all of the fundamental problems underlying real-time data management have been understood, although many have. More importantly, there is little or nothing in the way of production quality methods, designs, and software, with which to bring the technology that is understood to the field.

The remainder of this paper is organized as follows. Chapter 2 gives a high-level description of the services offered by database management systems. These are the benefits that might be made available in a cost effective way to real-time applications. Chapter 3 is a survey of the state of research in the most heavily researched aspect of real-time database management, namely concurrency control. Chapter 4 briefly describes other topics in the database literature that are of interest to real-time applications. Finally, Chapter 5 describes what remains to be done to make the practice of real-time data management more routine and cost-effective.

# 2. Benefits of Database Management Systems

It is worthwhile to recall the benefits of DBMS technology. Some benefits are not obvious, many have nothing to do with real-time computing *per se*, and all may get lost in the detailed discussions of subsequent chapters.

Database management systems provide central control of the data in a system. This centralization of control permits:

- Controlled elimination of redundancy. When data description is not centrally controlled, different applications or application segments maintain their own versions of the data. This not only wastes storage space, but, more importantly, may introduce inconsistencies as the various versions diverge.
- Maintenance of integrity constraints. The DBMS can protect the database from some classes of application errors: impossible data values (400 hours worked in a week), conflicting data (duplicate keys), etc.
- Publication of the data description as a resource.

The last item affects the application designers, rather than the applications. The database description, and the DBMS which realizes the described data, serves as a public interface between and among the *implementors* of large systems. Rather than have every pair of organizations and individuals involved in the construction and maintenance of a large system negotiate agreements to satisfy each other's need for information, the DBMS forms a common repository from which every application segment acquires its inputs and to which every segment deposits its results. The *enterprise* or *conceptual schema* [56] serves an information-hiding role and an information-publishing role which allow the parts of an application development team to work independently from each other.

A conceptual schema in the design of an application does not necessitate a DBMS in its implementation. But a good deal more work will be needed and a good deal more code will be generated if the those services are implemented by custom software. A fundamental question of this paper is: Can DBMS benefits be brought to real-time application construction at "off-the-shelf" prices?

One of the most important achievements of database technology is *data independence*, the separation of the functional and performance aspects of database interaction. An application specifies only the functional characteristics of its database interactions. With the needs of the complete collection of applications in view, the database administration staff specifies the storage structures and access paths that determine the performance characteristics of those applications. Of course, performance, i.e., timing, is a functional characteristic for real-time systems. It remains to be seen if conventional DBMS performance tuning is sufficient for real-time needs.

The technology of distributed database management has progressed to the point that many commercial DBMSs offer a degree of distribution, particularly on local area nets. This distribution is provided in a location-independent way. Like data independence, location independence separates the application software from performance-oriented decisions, in this case, data location within a network.

Database technology includes a form of fault tolerance and recovery. The faults involved are those which cause a loss of volatile main memory but not of secondary memory (disks).[2] Examples of such faults are power loss and operating system failure. The next chapter examines the utility of this notion of fault tolerance for real-time computing.

[2] When combined with checkpointing and sound risk management, e.g., offsite storage, these techniques also provide for recovery of secondary storage from more catastrophic failure.

# 3. Concurrency Control and Data Sharing

Database management systems traditionally provide facilities for the management of transactions. A transaction is an indivisible, or atomic, unit of work; a bank withdrawal or deposit and the sale of a seat on an aircraft are frequently cited examples of transactions. Battlefield management and stock arbitrage are often cited as examples of transaction processing systems with real-time constraints. Real-time transaction management adds the issue of timing constraint to conventional transaction management.

Conventional transaction management controls the concurrent execution of transactions to ensure that the effect of that execution on the database and its users is identical to the effect those transactions would have had, had they run in some serial (non-concurrent) order. The so-called "serializability" goal of transaction management is well established as an appropriate notion of correctness for schedules of interleaved transaction operations. Most (if not all) commercial DBMS rely on "two-phase locking" (2PL) concurrency controllers [16].

Real-time system designers and researchers are more likely to think in terms of tasks than transactions. Tasks and transactions are different computational concepts, and their similarities and differences affect how they are designed, implemented, and controlled.

- The task and the transaction abstractions are both units of work and therefore also units of design. In other words, a well designed task or transaction implements a well understood function or performs a well understood service.
- The task and the transaction abstractions are both threads of control; that is, they are the units of scheduling.
- All transactions are designed to terminate; many tasks are designed to be non-terminating.
- The transaction is a unit of recovery; the task is not. The atomic nature of a transaction requires that the effects of any failed transaction not be visible to any other transaction. Therefore, concurrently executing transactions must be isolated from each other; correct interleavings of transaction operations are serializable. Concurrently executing tasks are cooperative and inter-communicative; they are explicitly aware of each other and, as they request and perform services for each other, are explicitly *not* serializable.

The distinctions between tasks and transactions do not always hold, but are generally what a speaker has in mind when he or she uses the term *task* or *transaction*.

- The runtime behavior of a task is statically predictable; thus, the execution time of a task is calculable. The runtime behavior of a transaction is dynamic in that it depends upon the values it finds in the database. The execution time of a transaction is therefore difficult to predict accurately. When disk-resident data is involved, worst case estimates of transaction times may be wildly pessimistic. They are more likely to be simply unavailable.
- Tasks wait only for other tasks; transactions also wait for the completion of I/O events.
- Resources accessed by more than one task in a real-time system are few in number and generally known at design time. They take the form of critical regions or rendezvous containing procedures for updating shared data.

  The resources (database data granules) which *may be accessed* by multiple transactions of a transaction processing system are very many in number. The granules, which are in fact needed by any collection of concurrently executing transactions, will be relatively few in number. In other words, the database is much larger than its active, in-use subset, at any moment. There is generally a very low probability of conflict between transactions, a very low probability that any data granule will be needed by two concurrently executing transactions. However, the identity of the contested granules is not known until runtime.

As a consequence of these distinctions, critical regions and rendezvous, which always represent potential blocking, will be shorter than transactions, which represent potential blocking with low probability.

## 3.1. Consistent Schedules for Real-Time Applications

The popularity of serializability as a criterion for correctness of concurrent transaction execution is understandable. The argument for its correctness is straightforward and easy to grasp: if each transaction is correct, the effect of running any number of correct transactions in sequence must also be correct.[3] A few authors have ventured to define criteria for allowable interleavings of database operations that claim to allow non-serializable schedules. This section discusses a few of them.

Sha et al. [45] suggest a decomposition of the database into so-called *atomic data sets* with the property that the consistency of each atomic data set is defined without reference to anything outside of the data set, and that global database consistency is merely the non-interfering conjunction of these local consistency conditions. The authors [45] then suggest that the tasks of a real-time system be divided into *elementary transactions* with the property that each such transaction accesses only one atomic data set. Serializable executions of the elementary transactions may be non-serializable executions of the *compound transactions* or tasks from which they were derived. The essence of these ideas [45] is not "a generalization of serializability theory" but rather the recognition of the distinctions between the concepts of task and transaction.

The decomposition described by Sha et al. [45] results in decreased transaction lengths. Decreased transaction length, like decreased critical region size, results in fewer resource conflicts and shorter waiting times when conflicts do occur. It has been suggested [19] that the amount of blocking among a collection of transactions is proportional to the *square* of their lengths. Decreasing transaction length via decomposition is the most powerful tool a designer has for improving transaction performance.

The decomposition of the database into atomic data sets is done off-line by the software designers. This decomposition is essentially global to the system design and may not be stable with respect to system evolution. From an engineering point of view, what is needed is a method by which this decomposition or some other theory of transaction design, e.g.,

[3] The popularity of the 2PL algorithm—actually a constraint on the behavior of the applications and not an algorithm—stems from its ease of implementation.

the *transaction steps* of Garcia-Molina [18] or the *nested transactions* of Moss [36], can be carried out and managed in the presence of complex and evolving system requirements.

Lin [31, 57] suggests a decomposition of the database into *external* and *internal* objects and corresponding external and internal consistency constraints and external and internal transaction segments. (An external transaction segment is one which writes external objects that satisfy external constraints.) An external object "records the state of the real world."[4] Lin repeatedly stresses the opinion that the timeliness of external data is often more important than the consistency of the database. He does not give any method that exploits these insights nor a theory from which such a method might be derived. He does present some interesting examples, to which we will return.

I suggest that there is no substitute for serializability as a definition of correct interleavings of transaction operations. As is discussed in Section 3.4, real-time transaction schedulers produce serializable schedules that meet time constraints, a proper subset of the serializable schedules. As shown by Sha et al. [45], the application of the theory to practice, in particular, the identity of the transactions to be serialized, is not trivial. I suggest that if the programming agents to be controlled are not to be serialized *in some sense*, then they are not transactions. Lin remarks that "most real-time transactions are *cooperating* rather than *conflicting* with each other" [57]. In that case, their interactions are best understood in the theory of concurrent cooperating systems and parallel programming. There may be room for a theory of interactions of programs that are both cooperating and competitive, that are, for example, Ada tasks and Structured Query Language (SQL) transactions (see [11] and see also the transaction concept in Common APSE Interface Set-A (CAIS-A [8]), but the shape and the utility of such a theory are far from clear.

## 3.2. Temporal Consistency

The following example appears in Vrbsky et al. [57]:

> [Consider] a system in which robots use a database to recognize objects in front of them. . . . Suppose T1 is to recognize what is in front of a moving robot and T2 is to receive the current view. When the robot is far from the object, it will be difficult for T1 to identify the object. . . .[I]f T2 [receives] a closer picture of the object, instead of trying to resume the old recognition process, T1 may want to use the new picture from T2 to get a more effective result.

The scheduling decision that this example implies, the abort/restart of T1, would not be made by any conventional scheduling algorithm. The concept of "temporal consistency" suggests that the age of data be taken into account in making scheduling decisions. This is a new form of timing constraint, specifying bounds on the *start* time of one transaction relative

[4] This is a rather confused idea, as the purpose of any database is to record the state of the real world. Lin apparently has in mind the distinction between measurements of the world, e.g., sensor readings, and inferences about the world, e.g., track identities.

to the *stop* time of one or more transactions. [21] Temporal consistency can reference either the absolute age of the data read by a transaction, or the *dispersal* of those ages, the age of each data item relative to the age of every other data item in the read set of a transaction [51].

Liu and Song [51] apply temporal consistency *along with* serializability as a criterion for correctness. This is consistent with the observation that the correct operation interleavings for real-time transactions are those serializable interleavings which meet their timing constraints. No one has proposed a scheduler that uses temporal consistency in making scheduling decisions. (Liu and Song use it to judge the effectiveness of scheduling algorithms.)

Temporal consistency is used as the only criterion for correctness by the telemetry data acquisition functions of the Real Time Data System, part of the ground control software of the NASA shuttle system [37]. The *stuffer* (sic) routines organize "time homogeneous" buffers for evaluation by application tasks. These routines form a classic producer/consumer collection; their executions are serialized by design.

Temporal consistency is an interesting concept about which more can be learned. For example, no one has used temporal consistency in a manner that naturally requires the abort/restart of Lin's example (quoted above).

## 3.3. Imprecise Computation

As noted by Lin et al. [30], some algorithms, particularly the numeric algorithms, proceed by a method of successive approximation. They [30] propose to use such algorithms to trade accuracy for time. If "time runs out" (that is, a deadline is reached) before an algorithm computes a precise result, the larger system may be willing to use a prior approximation of the result.

Smith and Liu [47] have extended imprecise computation to the database realm. Their algorithms compute query answers incrementally. Hou et al. [23] present methods for computing approximations of aggregations (Count, Sum, etc.) with statistical techniques.

Strictly speaking, imprecise computation is not a database topic, as the computations are considered to produce answers, not to update shared data. Imprecise computation affects scheduling decisions [12], although the decisions are not specific to transaction scheduling. From an engineering point of view, the use of imprecise computations requires an understanding of the effect of imprecise answers on the application.

## 3.4. Real-Time Transaction Schedulers

I have argued that serializability is the correct notion of transaction concurrency control. In real-time applications, not every serializable schedule is acceptable; those which fail to satisfy timing constraints are incorrect. Researchers of real-time transaction schedulers ( [1], [2], [3], [6], [10], [22], [24], [35], [46], [48], [49], [50], [51], and [53]) have developed and analyzed a great many paradigms that consider timing constraints when making scheduling decisions.

*Priorities* can be *assigned* to real-time database transactions by many of the same strategies used to assign priorities to real-time tasks. As transactions are generally less predictable than tasks, priority assignment strategies using information about runtime behavior (e.g., execution time, needed resources, etc.) may not be feasible. Of the papers cited, only four ( [1], [2], [24] and [53]) consider priority assignment strategies. The authors of the other papers are content to assume priority is determined by some unexamined mechanism.

Transactions allow for *conflict resolution strategies* not available for tasks. When a task discovers another task in a contested critical region, it has little choice but to wait for the resident of the region to depart. A transaction owning a contested resource may be *aborted*, thereby freeing the resource. As transactions are units of recovery, transaction managers must have facilities to repair the effects of aborted transactions. This is not the case for task managers. Aborting a task in a critical region will generally leave a shared resource in a corrupt state.

The obvious modification of 2PL with priorities is as follows: When a transaction *T* requests a resource owned by transaction *U*, if the priority of *T* is not greater than that of *U*, then *T* is blocked, as in standard 2PL. If, however, *T*'s priority exceeds that of *U*, then *U* is aborted. This policy is discussed in the proceedings of three conferences ( [24], [1], [2]). A variation [1] considers the priority of the newly restarted instance of transaction *U*. In a least slack priority scheme, the new *U* has a higher priority than the old *U* and may have a priority higher than *T*. If it does, *U* is not aborted.

*Optimistic concurrency control* [25] exploits the low probability, under reasonable assumptions on transaction behavior, that any two concurrent transactions request the same database granule. It is a non-locking protocol: all database operations issued by transactions are performed when requested, although write requests do not immediately update the shared database but are deferred until such time as the transaction attempts to commit. When a transaction issues a commit request, the set of items read by the transaction is examined for any item written by some transaction that was committed since the committing transaction started. If such an item exists, the committing transaction is aborted; otherwise, it is committed. Since writes effectively occur at commit time, the serialization order selected by an optimistic concurrency controller is the order in which the transactions commit. In other words, the effect of a collection of transactions controlled by such a controller is the effect they would have had had each transaction executed, atomically, at its commit time.

Haritsa et al. [22] adapt optimism to the presence of priorities. Commit processing is modified as follows: if the committing transaction has written an item read by some concurrent transaction still in execution, the priorities of the two transactions are compared. If the committing transaction has the higher priority, the executing transaction is aborted. As that transaction would be aborted by standard optimistic protocols when it attempted to commit, the resources that it would have consumed are saved. If, however, the executing transaction has the higher priority, the committing transaction may abort itself or it may wait until the executing transaction commits, in hopes it will not be aborted by some higher priority transaction in the interim. Haritsa et al. [22] also present a compromise strategy, in which the committing transaction waits until fewer than half of the conflicting transactions have higher priority. Once that state is reached, remaining conflicting transactions are aborted, irrespective of their relative priorities, and the committing transaction is committed. The goal is avoiding the loss of work already accomplished for the committing transaction.

Timestamping [42] is another concurrency control method that avoids the use of locks. Every transaction is assigned a timestamp at the moment it enters the system. Those transactions that commit will be serializable in timestamp order, i.e., in the order in which they began. Any read or write operation that would, if executed, invalidate that ordering causes the requesting transaction to abort.

The order in which the set of active transactions arrived in the system has generally no relationship to their relative real-time priorities. It would appear that timestamping is not amenable to real-time scheduling.[5] Marzullo [35] presents a scheme for assigning timestamps, which he claims respects a static priority assignment. The rules are:

1. The timestamp assigned to a transaction is greater than the timestamp of any committed transaction.
2. The timestamp $s_i$ of transaction $t_i$ is set such that, for any active transaction $t_j$ with timestamp $s_j$, $s_i < s_j$ if and only if the priority of $t_i$ exceeds that of $t_j$.

But these rules do not appear to be sound. Consider that transaction $t_1$ with timestamp 3 is executing at the moment that transaction $t_2$, with a priority greater than that of $t_1$, arrives in the system. Transaction $t_2$ is given timestamp 2, in accordance with rule 2. Transaction $t_1$ then commits and transaction $t_3$, with a priority greater than that of $t_2$, arrives. Transaction $t_3$ cannot be given a timestamp that is simultaneously greater than 3, to satisfy rule 1, and less than 2, to satisfy rule 2.

*Multiversion concurrency control* [5] interprets write operations as the creation of new versions of the items (in contrast to the update-in-place semantics normally given to writing). Timestamps are used to return "appropriate" versions for read requests. Song and Liu [51] notice that, if sensor readings create new versions rather than update in place, the tasks servicing the sensors will never be blocked due to conflicts with the tasks reading the

[5] A variant of timestamping, called *time intervals* [38], may have a priority-driven derivative, but I have not found it in the literature.

sensors. They do performance studies to determine the extent of temporal inconsistency (see Section 3.2) introduced under differing workloads. As the purpose of multiple versions is precisely to allow for the reading of old data, the results in [51] should be interpreted as describing overload behavior and delineating overload thresholds under various workload assignments.

Lin and Son [32] present a complex protocol which shares features of optimistic (deferred writing, delayed selection of serialization order) and multiversion concurrency control. They do not present any performance figures, making it impossible to determine if the complexity of their algorithm is cost effective.

The priority ceiling protocol [46] is the only transaction scheduling system yet proposed that never aborts transactions. Interestingly, Marzullo [35] denies the existence of any priority-based scheduling mechanism that never aborts transactions. This is because he, and most other researchers, consider only systems in which priority inversions never occur.[6] The priority ceiling protocol seeks only to put a static, constant time bound, a bound that does not depend on runtime system behavior, on the length of any period of priority inversion.

The essential mechanism of priority ceiling protocols is the temporary promotion in priority of any transaction which holds a data granule to the highest priority of any transaction which may request the granule. Although Sha et al. [46] do not discuss of how this "highest priority" is determined, it must be determined statically, that is, from the design and implementation documents rather than from the dynamic system behavior. That implies that these priorities are determined not for the granule itself but for its type. Therefore, a transaction owning a particular granule, no matter its priority, blocks all transactions that *may* request granules of that *type*. The other protocols discussed deal only with actual, not potential, conflicts. The authors [46] claim that this is "the ‘insurance premium’ for preventing mutual deadlock and [ensuring] the block-at-most-once property," although it is more accurately the price paid for never aborting transactions. This price may be prohibitive when the size of a data granule is quite large, e.g., a page or a file. Under those conditions, it is likely that every data access potentially conflicts with every other data access, effectively serializing access to the database.

## 3.5. Scheduling I/O

Serialization of transactions is not the only scheduling process undertaken by a DBMS. A DBMS has control over the order in which I/O events occur and over the allocation and deallocation of buffers to be used by those events. The effect of deadlines and priorities on these aspects of database management have been studied only recently, notably by Carey et al. [10] and Abbott and Garcia-Molina [3]. These authors present simulation studies of various algorithms for scheduling disk I/O and managing buffers. They show that consid-

[6] A priority inversion occurs when a transaction must wait for actions, e.g., lock releases, of transactions with lower priority.

erable improvement in meeting timing constraints can be obtained by I/O schedulers which take those constraints into account in their scheduling decisions.

As noted, transaction concurrency control may defer writing to the database until the requesting transaction commits. This is a popular strategy, since it means that no repair work is necessary for failed transactions. Since all database writes occur after transaction commit, Abbot and Garcia-Molina [3] maintain that writes may be done "at leisure," even in real-time systems. This leads them to propose a novel I/O architecture that treats reads and writes asymmetrically.

## 3.6. Fault Tolerance and Failure Recovery

As noted at the start of this chapter, transaction atomicity requires the existence of transaction failure recovery in the transaction monitoring system. Nevertheless, none of the work previously discussed deals with issues of transaction failure recovery.[7] Of the simulation studies, only two ( [2] and [3]) mention the existence of failure recovery, and their purpose is to dismiss its impact on the simulation. The argument is that the log device is distinct from the database device; therefore, database I/O scheduling is independent of log I/O scheduling. This is not strictly correct.

All commercial DBMSs follow a write-ahead log policy. All database updates are recorded in the log before being recorded in the database. If this were not done, and if a system failure occurred after an update but before transaction completion, it would be impossible to roll the database, backward or forward, into a consistent state, i.e., a state in which the transaction had either completed or never started. Whereas database updates may be delayed until after transaction commitment, log writes may not be.[8]

The popularity of logging, as opposed to shadowing [34], for example, is due in part to the fact that, since logs may be removed from the system and saved indefinitely, they may serve functions beyond those of transaction recovery. Logs can be used as audit trails. When used with periodic checkpointing, logs can be used to recover from catastrophic media failure (head crashes, fire, etc).

Conventional database recovery is a form of "backward error recovery" [7]. The database is returned to a state reflecting none of the modifications made by computations in process, i.e., uncommitted, at the time of the failure. That state is an approximation of the system's knowledge of the world at the time of the failure. At the point of recovery of a real-time system, some of that information may be of no interest. Consider, for example, the failure of a telephone switching system. When such a system is recovered, it may not be necessary

[7] This statement holds despite the titles of [45] and [57].

[8] In [2] and [3], the log device is a serial, dedicated device which experiences no schedule delays. Therefore logging overhead is part of overhead per data item and need not be explicitly simulated. This is not an unrealistic scenario, but surely there are environments in which no such device is available.

to recover the connections active at the time of failure. But the system will not want to lose the billing information for those connections. Therefore, the system recovers to a state in which some of the previous information is retained and some is discarded. This is likely to be typical of real-time recovery.

I have found no work that has been done on real-time transaction failure recovery. Considering that failure recovery consumes substantial resources even in the absence of failure, this work will have substantial benefit. Many of the fundamental questions of failure recovery must be re-examined for real-time data management. The fundamental recommendation is that *transaction failure recovery must be integrated into the system design for fault tolerance*. Just as conventional transaction scheduling treats all transactions alike, conventional recovery treats all data and all failures alike. Real-time recovery must consider various classes of failures and the time and resources available for performing the recovery.

Although there is research work to be done in real-time transaction recovery, the central question is: How are the results of that research to be transferred into practice?

# 4. Database Technologies with Impacts on Real-Time

This chapter discusses aspects of database technology that may prove useful to real-time applications. The list is admittedly arbitrary. The selection criterion is roughly this: technologies are included which are likely to be useful in applications having real-time data management needs *and not* readily available commercially.

## 4.1. Main Memory Databases

During the 1980's a good deal of research investigated the effects of the availability of very large main memories. The earlier work ( [14], [15]) considered effects on query processing strategies, data structures, and failure recovery mechanisms when a substantial percentage of the database could fit into the DBMS buffer pool. Other researchers ( [27], [28], [43], [17], [40], [41]) assume the entire database can be made main memory resident. Li and Naughton [29] allow for multiple processors as well as for massive main memory.

Real-time processing environments are often constrained by external factors, but the trend is towards larger memories. Real-time databases can be very large, but there should be many cases for which a database that is fully or mostly resident in main memory is appropriate. The research cited above suggests that algorithms specialized for main memory databases offer considerably improved performance over conventional DBMS with very large buffer pools. However, the engineering work in determining cost-benefit tradeoffs is yet to be done.

Much of the work in main memory databases addresses recovery techniques. As I've argued above, recovery must be reconsidered for the real-time case, and hence real-time recovery of main memory must likewise be reconsidered.

## 4.2. Database Machines

Although there are usually good reasons to avoid specialized hardware in production systems, the performance constraints of real-time systems may be impossible to meet in conventional architectures. Research into DBMS-specific hardware architectures is quite well established [39] and has resulted in several commercial offerings, include Britton-Lee's Intelligent Database Machine (IDM) and Terradata's Database Computer (DBC). These products are disk-based associative memories, possibly unsuitable for the rugged environments of many real-time systems. There is at least one proposal [55], by the English firm Software Sciences, for a microprocessor (transputer) network-based database machine, DIOMEDES, specifically designed to operate under such conditions.

A database machine is used by applications in much the same way as a database server is used by its application clients. The server-client architecture is ubiquitous on local area networks. Real-time systems implemented in such networks, e.g., ship and aircraft systems, can exploit that architecture at no added risk. Database machine insertion is accomplished through the substitution of a special purpose DBMS machine for a general purpose database server. The effect on application engineering, on the general purpose host machines, is minimal.

The future of real-time database applications is likely to include a role for specialized database hardware.

## 4.3. Active Databases and Knowledge-Based Systems

Knowledge-based and expert systems are becoming increasingly common in real-time systems [37], [20]. Even if the deductive, inferential tasks of the system have lax timing constraints, the interface between the database and knowledge base paradigms must not impose excessive overhead. The database component may act as a scheduler for the expert system by monitoring the enabling conditions for rules.

"Rule processing" is an active area of database research.[9] The particular needs of real-time rule processing have had some attention [13]. The area may not yet be ready for transition into practice, but the pressure to do so is already evident.

[9] Six of the 37 papers in the 1990 ACM SIGMOD conference were classified under the Rule Processing heading.

# 5. Conclusions

The state of the art and practice of real-time data management is not sufficiently advanced to warrant the creation of standards. Although some of the necessary research has been done, there remain issues, e.g., transaction recovery, which are not fully understood. Further development of the technology is not in itself sufficient, as the engineering needed to make the technology practical in the field has not been done.

Two very different collections of engineering artifacts are needed. Real-time system designers need *theories* and *methods* based on those theories which they can use:

- in the design of real-time databases at both the conceptual and access path level
- in the design and decomposition of real-time transactions
- in the design and integration of real-time data recovery schemes

But there must also be some cost-effective means of *implementing* these designs. In the commercial world, these means are supplied by the commercial software houses through the commercial DBMSs that they market. Those systems are unsuitable for many real-time applications. They are very large, very powerful, general purpose systems that are very difficult to analyze. They offer neither priority scheduling nor flexible recovery. Although isolated, real-time use of commercial DBMS has been reported [44], it is not likely such usage will become widespread. How can the cost benefits of commercial software be exploited in the construction of real-time systems?

The software engineering community has been developing the concept of "reuse-based," "domain-specific," and "mega-programming" software engineering techniques. The database community has developed the concept of extensible, application-oriented database management systems [4], [9], [54], [33]. These two strands of research might be profitably combined and applied to the real-time data management problem.

DBMS toolkit and building block technology has been primarily concerned with extensibility to unusual data structures, with an eye toward CAD/CAM and other applications in which data objects can be very large (e.g., engineering diagrams) and in which transactions can be very long and require configuration management and version control rather than concurrency control. These are not of central concern in real-time applications. In building a real-time application-specific DBMS, it might be necessary to customize:

- storage handling and mapping functions (e.g., main or secondary storage schemes)
- concurrency control paradigms and schedulers
- buffer management and disk scheduling
- recovery schemes for various classes of faults and of data

It is not possible at this time to determine if there exist "best" or "universal" solutions to these problems. In real-time systems, it seems advisable to assume that for every solution, there is a problem which it does not fit. Building block technology specifically allows for different solutions in different applications. It also allows much more rapid introduction of technology into practice.

Many of the functions of a DBMS (e.g., scheduling and resource control) are also functions of an operating system. Real-time applications cannot afford duplication of these functions. A DBMS application written in Ada might find its execution controlled by the:

- operating system process scheduler
- Ada runtime task scheduler
- DBMS transaction scheduler
- DBMS resource (lock) manager
- operating system virtual memory manager
- DBMS buffer manager
- disk scheduler (supplied either by the DBMS or the operating system or, in the worst case, both)
- DBMS recovery (log) manager

Clearly, this situation is intolerable for real-time applications. The lesson to be learned is that database building blocks must be integrated with operating system and other runtime environment building blocks in order to avoid wasteful duplication.

Research on database building blocks for real-time applications carries substantial risk. In order to succeed, it may be necessary to:

- Discover or invent "the" real-time DBMS architecture, if such a thing exists.
- Determine a good packaging material for the building blocks. If the blocks are constructed directly from source code, can they be both general and efficient enough to be reused in real-time applications?
- Show how to compose performance (or complexity) information for the building blocks into performance information for the complete system.

It is not certain that these things can be done but these are among the problems confronting the application of "mega-programming" to any real-time domain.

# References

**1.** Abbott, R. and Garcia-Molina, H. Scheduling Real-time Transactions: a Performance Evaluation. Proceedings of the Fourteenth International Conference on Very Large Data Bases, Very Large Data Base Endowment, 1988, pp. 1-12.

**2.** Abbott, R. and Garcia-Molina, H. Scheduling Real-time Transactions with Disk Resident Data. Proceedings of the Fifteenth International Conference on Very Large Data Bases, Very Large Data Base Endowment, 1989, pp. 385-396.

**3.** Abbott, R. and Garcia-Molina, H. Scheduling I/O Requests with Deadlines: a Performance Evaluation. Proceedings 11th Real-Time Systems Symposium, December 1990, pp. 113-124.

**4.** Batory, D. S., Leung, T. Y., and Wise, T. E. "Implementation concepts for an extensible data model and data language". *ACM Transactions on Database Systems 13*, 3 (September 1988), 231-262.

**5.** Bernstein, P. and Goodman, N. "Multiversion Concurrency Control - Theory and Algorithms". *ACM Transactions on Database Systems 8*, 4 (December 1983), 465-484.

**6.** Buchman, A. P., McCarthy, D. R., Hsu, M., and Dayal, U. Time-Critical Database Scheduling: a Framework for Integrating Real-Time Scheduling and Concurrency Control. Proceedings of the Fifth International Conference on Data Engineering, 1989, pp. 470-480.

**7.** Burns, A. and Wellings, A.. *Real-Time Systems and their Programming Languages.* Addison-Wesley Publishing Company, Wokingham, England, 1990.

**8.** SofTech Inc. *Proposed Military Standard Common Ada Programming Support Environment (APSE) Interface Set (CAIS) (Revision A).* Naval Ocean Systems Center, as agent for Ada Joint Program Office, Waltham, MA, 1988.

**9.** Carey, M. J., Dewitt, D. J., Richardson, J. E., and Shektia, E.I. Object and File Management in the EXODUS Extensible Database System. Proceedings of the Twelfth International Converence on Very Large Data Bases, Very Large Data Base Endowment, August 1986, pp. 91-100.

**10.** Carey, M. J., Jauhar, R., and Livny, M. Priority in DBMS Resource Scheduling. Proceedings of the Fifteenth International Conference on Very Large Databases, 1989, pp. 397-410.

**11.** Chastek, Gary, Graham, Marc H., and Zelesnik, Gregory. Notes on Applications of the SQL Ada Module Description Language (SAMeDL). Tech. Rept. CMU/SEI-91-TR-12, Software Engineering Institute, June 1991.

**12.** Chung, J-Y. and Liu, J. W-S. Algorithms for Scheduling Periodic Jobs to Minimize Average Error. IEEE Real Time Systems Symposium, 1988, pp. 142-151.

**13.** Dayal et al. "The HiPAC Project: Combining Active Database and Timing Constraints". *SIGMOD Record 17*, 1 (March 1988), 51-70.

**14.** DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., and Wood, D. Implementation Techniques for Main Memory Database Systems. Proceedings of the ACM SIGMOD Conference on Management of Data, 1984, pp. 1-8.

**15.** Elhardt, K. and Bayer, R. "A Database Cache for High Performance and Fast Restart in Database Systems". *ACM Transactions on Database Systems 9*, 4 (December 1984), 503-525.

**16.** Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. "The notions of consistency and predicate locks in a database system". *Communications of the ACM 19*, 11 (1976), 624-633.

**17.** Fan, C. and Eich, M. H. Performance Analysis of MARS Logging, Checkpointing and Recovery. Proceedings of the 22$^{nd}$ Annual Hawaii International Conference on System Sciences, 1989, pp. 636-642.

**18.** Garcia-Molina, H. "Using Semantic Knowledge for Transaction Processing in a Distributed Database". *ACM Transactions on Database Systems 8*, 2 (June 1983), 186-213.

**19.** Gray, J. N., Homan, P., Korth, H., and Obermarck, R. A Straw Man Analysis of the Probability of Waiting and Deadlock. Tech. Rept. RJ3066, IBM Research Laboratory, February 1981.

**20.** Hall, D. L. and Llinas, J. Data Fusion and Multi-Sensor Correlation. Slides for a course.

**21.** Han, C-C. and Lin, Kwei-Jay. Scheduling Jobs with Temporal Consistency Constraints. Sixth IEEE Workship on Real-Time Operating Systems and Software, 1989, pp. 18-23.

**22.** Haritsa, J. R., Carey, M. J., and Livny, M. Dynamic Real-Time Optimistic Concurrency Control. Proceedings 11th Real-Time Systems Symposium, December 1990, pp. 94-103.

**23.** Hou, W-C., Ozsoyoglu, G., and Taneja, B.L. Processing Aggregate Relational Queries with Hard Time Constraints. Proceedings 1989 ACM SIGMOD International Conference on the Management of Data, June 1989, pp. 68-78.

**24.** Huang, J., Stankovic, J. A., Towsley, D., and Ramamritham, K. Experimental Evaluation of Real-Time Transaction Processing. IEEE Real-Time Systems Symposium, December 1989, pp. 144-153.

**25.** Kung, H. T. and Robinson, J. "On Optimistic Methods for Concurrency Control". *ACM Transactions on Database Systems 6*, 2 (June 1981), 213-226.

**26.** Lefler, M. Private Communication. GTE, Inc. private communication.

**27.** Lehman, T. J. and Carey, M. J. A Study of Index Structures for Main Memory Database Management Systems. Proceedings of the Twelfth International Converence on Very Large Data Bases, Very Large Data Base Endowment, August 1986, pp. 294-303.

**28.** Lehman, T. J. and Carey, M. J. Query Processing in Main Memory Database Management Systems. Proceedings of the 1986 ACM SIGMOD International Conference on the Management of Data, May 1986, pp. 239-250.

**29.** Li, Kai and Naughton, J. F. Multiprocessor Main Memeory Transaction Processing. Proceedings International Symposium on Databases in Parallel and Distributed Systems, IEEE, December 1988, pp. 177-187.

**30.** Lin, Kwei-Jay, Natarajan, S., and Liu, J. W-S. Imprecise Results: Utilizing Partial Computations in Real-Time Systems. Proceedings of the 1987 IEEE Real-Time Systems Symposium, 1987.

**31.** Lin, Kwei-Jay. Consistency Issues in Real-Time Database Systems. Proceedings of the Twenty-Second Annual Hawaii Internation Conference on System Science, 1989, pp. 654-661.

**32.** Lin, Y. and Son, Sang Hyuk. Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order. Proceedings 11th Real-Time Systems Symposium, December 1990, pp. 104-112.

**33.** Linneman, V., Kuespert, K., Dadam, P., Pistor, P., Erbe, R., Kemper, A., Suedkamp, N., Walch, G., and Wallrath, M. Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions. Proceedings of the Fourteenth International Conference on Very Large Data Bases, Very Large Data Base Endowment, 1988, pp. 294-305.

**34.** Lorie, R. A. "Physical Integrity in a Large Segmented Database". *ACM Transactions on Database Systems 2*, 1 (March 1977), 91-104.

**35.** Marzullo, K. Concurrency Control for Transactions with Priorities. Tech. Rept. TR 89-996, Dept of Computer Science, Cornell University, May 1989.

**36.** Moss, J. E. B. *Nested Transactions: An Approach to Reliable Distributed Computing*. Ph.D. Th., MIT Laboratory for Computer Science, March 1985.

**37.** Muratore, J. F., Heindel, T. A., Murphy, T. R., Rasmussen, A. N., and McFarland, R. Z. "Real-Time Data Acquisition at Mission Control". *Communications of the ACM 33*, 12 (December 1990), 19-31.

**38.** Noe, J. D. and Wagner, D. B. Measured Performance of Time Interval Concurrency Control Techniques. Proceedings of the Thirteenth International Converence on Very Large Data Bases, Very Large Data Base Endowment, 1987", pp. 359-367.

**39.** Ozkarahan, E.. *Database Machines and Database Management.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

**40.** Pucheral, P. and Thevenin, J-M. A Graph Based Data Structure for Efficient Implementation of Main Memory DBMS's. Tech. Rept. 978, INRIA, 1990.

**41.** Pucheral, P., Thevenin, J-M., and Valduriex, P. Efficient Main Memory Data Management using the DBGRAPH Storage Model. INRIA, 1991.

**42.** Reed, D. "Implementing atomic actions on decentralized data". *ACM Transactions on Computer Systems 1*, 1 (February 1983).

**43.** Salem, K. and Garcia-Molina, H. Checkpointing Memory-Resident Databases. Proceedings Fifth International Conference on Data Engineering, IEEE, February 1989, pp. 452-462.

**44.** Schlunk, R. W. "Use of relational database management systems with realtime process data". *Journal A 29*, 2 (June 1988), pp. 17-23.

**45.** Sha, L., Lehocsky, J. P., and Jensen, E. D. "Modular Concurrency Control and Failure Recovery". *IEEE Transactions on Computers 37*, 2 (February 1988), 146-159.

**46.** Sha, L., Rajkumar, R., Son, Sang Hyuk, and Chang, C-H. A Real-Time Locking Protocol. .

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

**48.** Son, Sang Hyuk. "Semantic information and consistency in distributed realtime systems". *Information and Software Technology 30*, 7 (September 1988), 443-449.

**49.** Son, Sang Hyuk. "Recovery in main memory database systems for engineering design applications". *Information and Software Technology 31*, 2 (March 1989), 85-90.

**50.** Son, Sang Hyuk. Scheduling Real-Time Transactions. Proceedings EuroMicro ’90 Workshop on Real Time, 1990, pp. 25-32.

**51.** Song, X. and Liu, J. W. S. Performance of multiversion concurrency control algorithms in maintaining temporal consistency. .

**52.** Stankovic, J. A. "Misconceptions about real-time computing". *Computer 21*, 10 (October 1988), 10-19.

**53.** Stankovic, J. A. and Zhao, Wei. "On Real-Time Transactions". *SIGMOD Record 17*, 1 (March 1988), 4-18.

**54.** Stonebraker, M. and Rowe, L. A. The POSTGRES Papers. Tech. Rept. UCB/ERL M86/85, Electronics Research Laboratory, UC Berkley, November 1986.

**55.** Tillman, P. R., Giles, R., and Wakley, I. Multi-Level Parallel Processing for Very High Performance Real-Time Database Management. Proceedings AGARD Conference Software Engineering and its Application to Avionics, NATO Advisory Group for Aerospace Reserch and Development, April 1988, pp. 35.1-35.11.

**56.** Tsichritzis, D. and Klug, A. (eds). "The ANSI/X3/SPARC DBMS framework report of the study group on database management systems". *Information Systems 3 3* (1978), 173-191.

**57.** Vrbsky, S. V. and Lin, Kwei-Jay. Recovering Imprecise Transactions with Real-Time Constraints. Seventh Symposium on Reliable Distributed Systems, October 1988, pp. 185-193.

**58.** Whalley, D. Private Communication. Dowy SEMA, Inc, private communication.

# Table of Contents