MRDB: A Multi-user Real-Time Database Manager for Time-Critical Applications

Sang H. Son
Robert C. Beckinger

Computer Science Report No. TR-91-13
May 21, 1991

MRDB: A Multi-user Real-Time Database Manager for Time-Critical Applications

Sang H. Son
Robert C. Beckinger

Computer Science Department
University of Virginia
Charlottesville, VA 22903

Abstract

The design and implementation of real-time database systems presents many new and challenging problems. Compared with traditional databases, database systems for time-critical applications have the distinct feature of satisfying timing constraints associated with transactions. Transactions in real-time database systems should be scheduled considering both data consistency and timing constraints. In addition, a real-time database must adapt to changes in the operating environment and guarantee the completion of critical tasks. The effects of scheduling decisions, and concurrency control mechanisms on real-time database systems have typically been demonstrated in a simulated environment. In this paper we present a functional real-time relational database manager, called MRDB, which provides an operational platform for research in real-time database issues. Current research issues involving the development of run-time estimates for use in scheduling decisions, temporal consistency characteristics, and our efforts in using these are also discussed.

This work was supported in part by ONR under contract N00014-88-K-0245, by CIT contract # CIT-INF-90-011, and by IBM Federal Systems Division.

1. Introduction

As computers become essential parts of real-time systems, real-time computing is emerging as an important discipline and an open research area in computer science and engineering [Stan88]. The growing importance of real-time computing in a diverse number of applications such as defense systems, industrial automation, aerospace and nuclear reactor control, has resulted in an increased effort in this area. Very few conventional database systems allow users to specify temporal constraints or ensure that the system meets those constraints. Interest in this new application domain is growing in the database community. Further evidence of the rising importance and the rapid growth of research in the real-time database field can be seen in the large number of research results which have recently appeared in the literature [Abb88, Abb89, Abb90, Sha88, Sha91, Son88, Son89, Son90a, Son90b, Son90c, Son91].

Real-time database systems (RTDBS) can be useful for applications which are both data intensive and subject to real-time constraints. Appropriate methods and techniques for designing and implementing database systems that take timing constraints into account are playing an ever increasing role in determining the success or failure of real-time systems. In recent workshops [IEEE90, ONR90], developers of real-time systems have pointed to the need for basic research in database systems that satisfy timing constraint requirements in collecting, updating, and retrieving shared data.

The distinct feature of real-time database systems, as compared to traditional databases, is the requirement to satisfy the temporal aspects associated with transactions. Time is the key to be considered in RTDBS. The correctness of the system depends not only on the logical results but also on the time within which the results are produced. The temporal aspects associated with a transaction include timing constraints and a temporal consistency component. Transactions must be scheduled in such a way that they can be completed before their corresponding deadline expires, and the information requested returned according to the specified temporal consistency requirements. For example, both the update and retrieval transactions on the tracking data of a missile must be processed within the given deadlines: otherwise, the information provided could be of little value. In such a system, transaction processing must satisfy not only the database

consistency requirements but also the timing constraints associated with the transaction [Son90b].

A real-time database system has many similarities with conventional database management systems and with conventional real-time systems. The RTDBS lies as the interface between the two type of systems, and is not quite the same as either one of the two. A RTDBS must process transactions and guarantee that database consistency is not violated just as in a conventional database system. Conventional database systems, however, do not stress the notion of time constraints or deadlines with respect to transactions. Individual temporal constraints are not taken into account when making scheduling decisions. The performance goal for conventional database systems is usually expressed in terms of minimizing average response times instead of constraints on individual transactions.

Conventional real-time systems do take transaction temporal specifications into account, but ignore data consistency issues. Real-time systems also typically work with processes which have predictable resource requirements, to include data requirements. A RTDBS tends to make unpredictable data accesses. This exasperates the scheduling problem, and highlights another difference between a conventional real-time system and a real-time database system. The conventional real-time system attempts to ensure that no temporal constraints are violated. In a RTDBS it is impossible to guarantee all temporal constraints because of the unpredictable random data accesses, so the system must strive to minimize the number of constraints which are violated [Abb89, Son90b, Son90c].

State-of-the-art database systems are typically not used in real-time applications due to two major inadequacies: lack of predictability and poor performance [Son88, Son89]. Current database systems do not schedule their transactions to meet response requirements and they commonly lock data objects to assure the consistency of the database. The problem is that locks and time-driven scheduling are incompatible. Low priority transactions can and will block higher priority transactions leading to priority inversions and response requirement failures. Recently, a considerable research effort has been focused on real-time database scheduling and data consistency control mechanisms. The integration of the two in real-time database systems is not

trivial, because all existing concurrency control methods synchronize concurrent data accesses by the combination of two measures; block and rollback, both of which create barriers for time-critical scheduling. Concurrency control mechanisms such as Priority Inheritance, Priority Ceiling Protocol, and Conditional Restart have been studied and implemented in an attempt to manage the integration of real-time scheduling and data consistency requirements in real-time databases [Abb89, Car89, Hua90, Lin90, Sha91]. They are typically compatible with time-driven scheduling, and meet both the required system response predictability and temporal consistency.

The design and evaluation of a RTDBS presents challenging problems. In this paper we describe a functional multi-user real-time relational database system (MRDB), which we have developed for exploring real-time database issues. In addition, we examine issues involving the development of credible run-time estimates for use in real-time database scheduling decisions, the integration of data and temporal functionality, and the use of temporal consistency specifications in performing database operations. The remainder of the paper is organized as follows. Section 2 provides the reader with information on temporal constraint issues and temporal functionality which were sought in the design of MRDB. Section 3 describes our multi-user real-time relational database (MRDB) and the environment in which it operates. Section 4 presents real-time scheduling policies that are implemented in MRDB, our derivation and use of run-time estimates and some initial run-time performance measurements. Finally, section 5 summarizes the work conducted and the areas of future work.

2. Temporal Functionality and Issues

One of the major goals in designing real-time database systems is to meet timing constraints. It is also one of the major problems when designing a real-time scheduler which attempts to minimize the probability of transactions failing to meet their respective deadlines. Various approaches have been investigated or designed in developing systems which attempt to achieve this goal. The designers of CASE-DB [Ozso90] used an iterative evaluation technique coupled with a *risk* probability attribute in an attempt to provide as much information as possible within a given deadline. The priority ceiling protocol, which was initially developed as a task scheduling protocol

for real-time operating systems, has been extended for use in RTDB systems [Sha91]. It is based on a two-phase locking protocol and employs blocking, versus rollback, in an attempt to minimize the number of transactions which fail to meet their deadline.

Approaches such as these attempt to make scheduling decisions based mainly on transaction attributes such as priority, release time and deadline. These transaction characteristics are critical pieces in the scheduling puzzle, but they are not the only attributes available for use in solving the problem. One key attribute absent from most scheduling decisions is a viable transaction *run-time estimate*. Numerous research efforts have explored the possibility of using run-time estimates in the scheduling decision process. Run-time estimates have been used in workload policies, priority assignment policies, conflict resolution policies and IO scheduling policies. These run-time estimates have typically been model-driven. The results derived have shown that run-time estimates are a credible option for use in scheduling decisions. However, the derivation and use of run-time estimates in a functional real-time database has not been explored extensively. Schedulers which do not incorporate run-time estimates into account are failing to use a key attribute which can simplify the scheduling decision. Scheduling decisions which do not take computation requirements into account allow such occurrences as processor time to be expended upon transactions which cannot meet deadline criteria.

If the real-time database scheduler can be provided with an estimate of transaction execution time, that information can be used in determining which transaction is closest to missing a deadline, and hence should be given higher priority, or which transaction can be delayed without risking violation of their timing constraints. In addition, run-time estimates can be used by the scheduler to initially screen transactions to determine eligibility. All transactions with feasible deadlines (release time plus run-time estimate less than deadline time) remain in the system and are eligible for service, while all ineligible transactions are aborted.

We sought predictability and accuracy in exploring the feasibility of using run-time estimates in scheduling decisions for real-time database systems. Without a predictable and accurate run-time estimate, little can be gained in the scheduling decision cycle, while leaving the system susceptible to unpredictable behavior. That is not to say that run-time estimates have to be correct 100% of the

time, since the typical real-time database performance goal is usually expressed in terms of minimizing missed deadlines, not guaranteeing no missed deadlines. However, because of their serious impact on scheduling decisions, the run-time estimates must be both predictable and reliable.

Often a significant portion of a real-time database is highly perishable in the sense that it has value only if it is used in time. In addition to deadlines, therefore, other kinds of temporal information should be associated with data as well as transactions in a real-time database system. For example, each sensor input could be indexed by the time at which it was taken. Once entered in the database, data may become out-of-date if it is not updated within a certain period of time. To quantify this notion of *age*, data may be associated with a valid interval. The valid interval indicates the time interval after the most recent updating of a data object during which a transaction may access it with 100% degree of accuracy. What occurs when a transaction attempts to access a data object outside of its valid interval is dependent upon the semantics of data objects and the particular implementation. For some data objects, for instance, reading it out of its valid interval would result in 0% accurate values. In general, each data object can be associated with a *validity curve* that represents its degree of validity with respect to the time elapsed after the data object was last modified. The system can compute the validity of data objects at the given time, provided the time of last modification and its validity curve [Liu88, Son91].

A real-time transaction should include its temporal consistency requirement which specifies the validity of data values accessed by the transaction. For example, if the temporal consistency requirement is 10, it indicates that data objects accessed by the transaction cannot be *older* than 10 time units relative to the start time of the transaction. This temporal consistency requirement can be specified as either hard or soft, just as deadlines are. If it is hard, an attempt to read an invalid data object (i.e., out of its valid interval) will cause the transaction to be aborted.

While a deadline can be thought of as providing a time interval as a constraint in the future, temporal consistency specifies a temporal window as a constraint in the past. As long as the temporal consistency requirement of a transaction can be satisfied, the system must be able to

provide an answer using available (may not be up-to-date) information. The answer may change as valid intervals change with time. In a distributed database system, sensor readings may not be reflected to the database at the same time, and may not be reflected consistently due to the delays in processing and communication. A temporal data model for real-time database systems must therefore be able to accommodate the information that is partial and out-of-date. One of the aspects that distinguishes a temporal data model for a real-time database systems from that of conventional database systems is that values in a real-time database system are not necessarily correct all the time, and hence the system must be selective in interpreting data values [Son90c].

Another design goal of real-time database systems is to enhance the temporal functionality associated with the data stored within the database. Temporal information has been stored in conventional databases for many years; accounting and payroll systems are typical examples. In these systems the attributes involving time are manipulated solely by the application programs. None of these systems interpret temporal domains when deriving new relations or extracting data. Most conventional database systems represent the state of an enterprise at a single moment of time. Although the contents of the database continue to change as new information is added, these changes have typically been viewed as modifications to the state, with the old, out-of-date data being deleted from the database. The new state also does not necessarily reflect the current status of the real world, since changes to the database always lag behind changes in the real world [Sno87].

3. Implementation of MRDB and Assumptions

MRDB is a functionally complete relational database manager. It offers not only a functionally complete set of relational operators such as *project, select, join, union* and *set difference*, and aggregate operators such as *max, min, avg, count* and *sum*, but also necessary relation operators such as *create, insert, update, delete, rename, compress, extract, import, export, sort*, and *print*. These operators give the user a fair amount of relational power and convenience for managing the database.

MRDB is a multi-user real-time relational database system whose origins can be traced back to SDB (Simple Relational Database) [Bec90]. SDB is a single-user relational database system, and

hence the code is not necessarily re-entrant. Numerous modifications had to be made to provide a multi-user environment with temporal functionality. MRDB is designed along the traditional client-server paradigm. It has a multiple-threaded server that is capable of accepting MRDB commands from multiple client sites. MRDB was designed with the goal of providing a temporal platform for conducting research on real-time database issues. It allow us to analyze real-time database mechanisms in an operational environment. This is a major and natural step forward from performance analysis conducted in a simulated real-time database environment. Simulated environments are not a substitute for functional systems. They fail to account for all factors found in an operational system, and tend to be more subjective in the sense that system parameters can be readily modified. An operational system cannot be modified to fit the real-time mechanism being analyzed. The results derived from an operational real-time database system provide us with a set of more realistic performance measurements.

The MRDB server is the heart of the database management system. It is responsible for receiving and acting on requests from multiple clients, and returning desired information to the clients. The server contains an infinite loop that accepts high-level database requests (e.g., create, union, insert) from multiple clients. The requests come in as packets. The MRDB system provides two different types of packets: *call* packets and *return* packets. The call packet is created by the client and is the database transaction. The call packet contains all the information that the server needs to carry out the desired database access operation, to include the timing constraint and temporal consistency specifications associated with the transaction. Clients are able to specify timing constraints and temporal consistency specifications for each transaction submitted to the server thread. A different timing constraint can be specified for each transaction submitted, or the client can allow the timing constraint to default to a system deadline previously established. The MRDB client thread passes the call packet forward to the MRDB server. The server performs some preprocessing and then forwards the packet to the MRDB scheduler.

The MRDB scheduler uses a run-time estimate evaluation technique to determine if the system can provide the client with the information requested within the timing constraint specified. The MRDB server will spin-off a separate MRDB thread to execute the transaction if the scheduler

makes the determination that a transaction can be computed within the given deadline. The client will be informed, and no thread spun-off if the MRDB scheduler determines that the result cannot be completed within the specified timing constraint. The thread will execute until completion and then forward the call packet back to the client. The client thread will process the return packet accordingly. A transaction is not preempted by the MRDB thread even if the determination has been made that a deadline is missed. The fact that a transaction has missed its deadline will be reported to the client, along with the results of the transaction.

MRDB also associates a temporal *'valid time'* attribute with each relation created in MRDB. This is inherent to the system, requiring no client involvement. The temporal attribute is attached to each tuple of a relation and is comparable to a timestamp that represents the *valid time* that the stored information models reality. The client cannot set or modify the values associated with the *valid time* attribute. However, this attribute can be manipulated for use in specifying transaction temporal consistency requirements. For example,

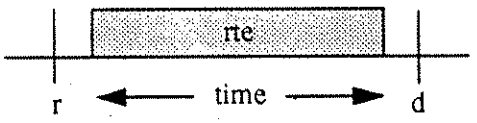
select trk_num from trackfile where *valid time* < 1

will return only the track numbers (trk_num) of tuples inserted or updated in the database relation (trackfile) within the last second of the transaction release time. Track numbers with *valid time* attribute values older than a second are active within the relation, but do not satisfy the temporal consistency requirement specified. MRDB also allows users of the system to manipulate the valid time attribute in output displays and in creating and manipulating relations that are similar to those found in *historical* temporal database systems [Sno87]. A relation without the temporal attribute *valid time* attribute can be formed by *projecting* or *selecting* attributes other than *valid time* into a new relation. The MRDB system will process such relations without attaching any temporal meaning to them.

The MRDB system employs a *strict* two phase locking (2PL) protocol for concurrency control [Ber87]. The *strict* locking protocol was selected for concurrency control because of its prevalence in commercial applications system and because of its desirable characteristic of being recoverable

and avoiding cascading aborts. Furthermore, abort can be implemented by restoring before images. Numerous conflict resolution policies such as High Priority, Priority Inheritance, Priority Ceiling Protocol and Conditional Priority Inheritance have been studied extensively in conjunction with a locking protocol environment [Abb89, Hua90, Lin89, Sha88]. The results indicate that such conflict resolution policies are compatible with time-driven scheduling, and meet both the required goals of system response predictability and temporal consistency. The area of conflict resolution policies, a significant area with respect to the scheduling of transactions in a manner which minimizes missed deadlines, is an ongoing area of research within MRDB, and is not addressed further in this paper.

A MRDB transaction is characterized by its timing constraints and its computation requirements. The timing constraints are a release time *'r'* and deadline *'d'*. The release time is the time associated with the transmittal of the transaction by a client site. A computation requirement is represented by a run-time estimate *'rte'* which approximates the amount of computation, IO, and communication costs associated with processing a transaction. The deadline corresponds to the client specified timing constraint.



The release time and deadline are known to the MRDB scheduler when a transaction arrives. The computation requirements are calculated based on the operation being performed and the physical characteristics of the data involved. This information is made available prior to the scheduling decision being made. We feel it is viable to estimate the execution time of a transaction without having prior knowledge of the exact data access pattern of a transaction.

The goal of our system is to minimize the number of transactions that miss their deadlines, i.e., that finish after time *'d'*. If transactions can miss deadlines, one must address the issue as to what happens to transactions that have already missed their deadlines but have not yet finished. There

are two alternatives. One is to assume that a transaction that has missed its deadline can be aborted. This may be reasonable where the value of a transaction is dependent on the timeliness of the return response. For example, suppose that a transaction is submitted to update the ballistic path of a projectile based on a radar sensing. If the deadline is missed, it may be more desirable not to perform the operation of updating the ballistic path, but instead to re-submit the update request based on a newer sensor reading. The conditions that led to the triggering of the transaction may have changed. The initiator of the transaction may be better served if the transaction is re-submitted.

A second option is to assume that all transactions must eventually be completed, regardless of whether they have missed their deadlines. This may be a correct approach in an application such as banking where a customer would rather have his financial transaction done late rather than not at all. If the decision is made to process the transaction, there is still the issue of the priority of tardy transactions with respect to other transactions in the system. Transactions which cannot meet their deadlines could receive a higher priority as their lateness increases, or they could be postponed to a later more convenient time.

The MRDB implementation decision was a combination of the two approaches. When a transaction enters the system, a determination is made as to whether a transaction can be executed within the temporal constraint associated with it. If the transaction cannot meet its deadline, it is aborted. This has the nice quality of not allowing computation time to be expended on transactions which cannot physically meet their deadlines. To allow such transactions into the system can adversely affect overall system performance especially during high load periods. Aborting a few late transactions helps all other transactions meet their deadlines, by eliminating the competition for resources by tardy transactions. Once a transaction has been accepted for processing, it is executed to completion, regardless as to whether or not a deadline has been met. This approach was adopted as a means of validating the run-time estimates derived by the scheduler.

The MRDB system has been developed on Sun workstations under the Unix operating system. MRDB is written in C and designed to operate across a local area network, with multiple client

nodes accessing the centralized database maintained by the system. MRDB was designed for Unix because of the prevalence of the operating system. MRDB has the nice property of being readily ported to other Unix sites interested in real-time database research and issues. The argument can be made that real-time database operations need to be coherent with the operating system, because correct functioning and timing behavior of database control algorithms depend on the services of the underlying operating system. We have found, through our performance analysis of MRDB, that dedicated resources in an environment such as above can provide predictable, analyzable, and meaningful results.

4. MRDB Scheduling Policies and Run-Time Estimates

The MRDB scheduling algorithms have three components: a policy to determine which tasks are eligible for service, a policy for assigning priorities to tasks, and a conflict resolution policy. Only the first two policies are explored in the remainder of this paper.

4.1 Scheduling Policies

The MRDB scheduler is invoked whenever a transaction enters the system or terminates. The scheduler can also be invoked to resolve contention (for either the CPU or data) when conflicts occur between transactions. The first action of the scheduler is to divide the set of ready transactions into two categories, those transactions that are capable of meeting their temporal constraints (eligible) and those transactions that cannot meet their temporal constraints (ineligible). All ineligible transactions are aborted and the MRDB client is informed of the decision. Eligible transactions remain in the system and are eligible for further processing. This approach differs from a Not Tardy [Abb89, Abb90] policy which accepts transactions that are currently not late, but may be in a position where it is physically impossible to make their deadlines. Only those transactions with Feasible Deadlines are considered to be eligible. A transaction has a feasible deadline if its deadline is less than or equal to the current time plus its run-time estimate.

$$\text{current time } (\mathbf{t}) + \text{ run-time estimate}(\mathbf{rte}) \leq \text{ deadline } (\mathbf{d})$$

In other words, based on the run-time estimate, there is enough time to complete the transaction before its deadline. This policy can be adapted to account for the amount of service time a transaction has already received. The modified policy would be as follows:

$$\text{current time } (\mathbf{t}) + \text{ run-time estimate}(\mathbf{rte}) - \mathbf{p} \leq \text{deadline } (\mathbf{d})$$

Where '*p*' equals the amount of service time a transaction has accumulated. This modified policy allows transactions to be screened for eligibility during the course of execution. Transactions that have been blocked, due to either data or CPU contention, could be re-evaluated to determine if they are still capable of meeting their temporal constraint. Note that the success of both of these policies is contingent on the accuracy of the run-time estimate. Erroneous run-time estimates which over-estimate the actual computational requirements will cause transactions to be aborted needlessly. Low estimates can degrade system performance by allowing transactions, which in reality cannot meet temporal constraints, to compete for system resources among transactions which are trying to meet deadlines.

There are many ways for assigning priorities to real-time tasks. Three policies extensively studied by earlier researchers include *First Come First Serve* (FCFS), *Earliest Deadline* (ED) and *Least Slack.*(LS) [Abb88, Abb89]. The primary weakness of FCFS is that it does not make use of deadline information. It discriminates against a newly arrived task with an urgent deadline in favor of an older task which may not have such an urgent deadline. The ED policy has shown itself to be effective in certain applications, but it fails to take into account the run-time estimates. The LS priority assignment policy was adopted for MRDB. The slack time for a transaction is an estimate of how long we can delay the execution of a transaction and still meet its deadline. It is computed by subtracting the current time plus the run-time estimate from the deadline of the transaction.

$$\text{Slack } (\mathbf{s}) = \text{deadline } (\mathbf{d}) - (\text{current time } (\mathbf{t}) + \text{ run-time estimate } (\mathbf{rte}))$$

The smaller the slack, the higher the priority. A negative slack time is an estimate that it is

physically impossible for the transaction to meet its deadline. This priority assignment policy does not take the amount of prior service time into account. The assignment of priority is static, occurring once when the transaction enters the system. The priority computed at that time remains with the transaction throughout its execution life. A continuous LS policy could be used which does take service time into account. The continuous evaluation of priorities causes the LS of all active transaction to be recomputed whenever there is contention for the CPU or data. This continuous evaluation can lead to degraded performance as shown in the simple example of Figure 1. The example gives the parameters for three transactions, shows the LS computations for both continuous and static versions, and plots their CPU usage based on those priority assignments. The problem of continuous LS is displayed with the arrival of transaction T_3 at time 2.5. The arrival of

Given:

Transaction	r	rte	d
T_1	0	3	4
T_2	1	2	5
T_3	2.5	1	6

Time	Continuous LS s = d-(t+rte-p)	Static LS s = d-(t+rte)
0	$S_1 = 4 - (0 + 3 - 0) = 1$	$S_1 = 4 - (0 + 3) = 1$
1	$S_1 = 4 - (1 + 3 - 1) = 1$ $S_2 = 5 - (1 + 2 - 0) = 2$	$S_1 = 1$ $S_2 = 5 - (1 + 2) = 2$
2.5	$S_1 = 4 - (2.5 + 3 - 2.5) = 1$ $S_2 = 5 - (2.5 + 2 - 0) = 0.5$ $S_3 = 6 - (2.5 + 1 - 0) = 2.5$	$S_1 = 1$ $S_2 = 2$ $S_3 = 6 - (2.5 + 1) = 2.5$

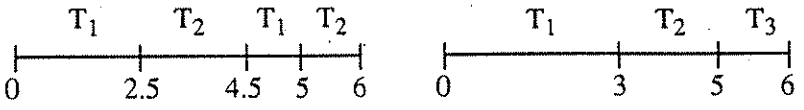


Figure 1. Continuous LS vs Static LS

T_3 causes the priority of T_2 to become higher than T_2, resulting in T_2 gaining control of the CPU, and T_1 being blocked. This causes all three transactions to miss their deadlines. The static LS version allows all three to meet their deadlines.

A negative slack time could occur if a transaction has already missed its deadline or is about to miss its deadline. The possibility of a negative slack time does not exist if a feasible deadline eligibility screening policy is implemented, and the LS priority assignment policy is static. The initial screening conducted to determine eligibility will eliminate any transaction which cannot physically meet their deadlines and the static LS priority will prevent the slack associated with an eligible transaction from ever becoming negative. The current MRDB versions uses static LS as the means of assigning priorities to transaction for scheduling, in an attempt to expedite those transactions which can least afford to be delayed.

4.2 Run-Time Estimates

Conventional real-time systems typically deal with processes which have predictable resource requirements. These predictable requirements allow for a static evaluation of computation costs. Real-time databases normally deal with transactions which have unpredictable resource requirements. The random nature of such data accesses complicates the scheduling process in real-time database systems. A considerable amount of research effort has focused on real-time database scheduling issues and the use of run-time estimates. The use of run-time estimates in scheduling decisions have been examined in workload screening, priority assignment, conflict resolution and IO scheduling policies. The results from the research conducted to date have indicated that run-time estimates are a viable option for improving scheduling decisions [Abb89, Son91]. The fact that critical information such as run-time costs can improve scheduling decisions and subsequently overall system performance is quite intuitive. However, the derivation of run-time estimates have typically been model driven. The derivation and use of run-time estimates in a functional real-time database has not been appropriately explored.

One of the goals in the design of MRDB was to derive credible run-time estimates and to integrate those estimates in scheduling decisions. The approach we used was to exploit the physical

characteristics of the data (such as attribute types, number of attributes in a relation, and the numbers of tuples in a relation) being manipulated, along with the type of database operation being performed (such as union, set difference and project), in an attempt to derive credible run-time estimates. While the arrival and types of transactions entering the system and the data which they access may be random, the computation steps involved in providing the appropriate response are not unpredictable. The steps required to execute any MRDB command is static in nature, and in a simplified outlook, only the number of iterations involved is dynamic.

The dynamic nature of the computation is dependent on the number and types of attributes involved, along with the number of tuples which constitute a relation. For example, the run-time cost for selecting values from a relation consisting of only a single tuple is minimal. It consists of basic start-up costs (such as transmitting the command, preprocessing, opening of relations, and reading in the data from disk), the actual computation cost in selecting that single value, and basic shut-down operations (such as providing the transaction results). The run-time cost for selecting the same set of values from a relation of five hundred entails the same basic costs associated with opening and closing operations for a single tuple relation, only the computation costs increase in relation to the number of tuples that have to be processed.

Other factors such as system load and data conflicts do not affect the run-time costs associated with a given transaction. Such factors only increase the competition for system resources, such as the CPU and IO access. For example, given the cost for selecting values from a given relation is '2' time units. If half that time is consumed, and that select operation is subsequently blocked by a higher priority transaction whether it be for CPU or data contention reasons, it will still require '1' time unit to complete once it becomes unblocked.

With this approach, we ran numerous performance measurements tests to capture the run-time costs. The results indicated that viable run-time estimates could be derived based on the physical characteristics of the data being manipulated and the operation being performed. The results which follow are a small extract from those numerous run-time cost analysis experiments. The results are based on database operations performed on relations of the format displayed in Figure 2. This

relation represents the track data generated by the Interim Battle Group Tactical Trainer, for an outer air battle scenario being used at the Naval Ocean Systems Center [But90].

Attribute Name	Type	Meaning
trk_num	integer	track number
lat_track	real	latitude of track
long_track	real	longitude of track
bearing	real	bearing from data link ref point
dep_high	real	depth or height of platform
lat_dlrp	real	latitude of data link ref point
long_drlp	real	longitude of data link ref point
platform_type	char	type of platform
cat	char	category of platform
time	integer	greenwich mean time
trkqa	integer	confidence of measurements
lat_tdir	char	latitude direction
long_tdir	char	longitude direction
course	real	bearing minus data link ref point
speed	real	speed of platform
range	real	range from ref pt in nautical miles
nuclear	char	nuclear classification of platform

Figure 2. Attributes of the track relation

4.3 Performance Results

The results of run-time estimate performance measurements for four basic MRDB commands (project, select, union, set difference) operating on relations displayed above are given in Figure 3, and graphically displayed in Figure 4. The x-axis of Figure 4 is in total tuples processed by the operation. The y-axis is the total elapsed time from the start of the operation until the final result is received at the client node. The performance measurements were conducted in an attempt to isolate the cost factors attributable to the operations performed, and the size of the data processed (measured by the number of tuples in the relations). The operations were initiated from a separate client node, transmitted to the server node, and the appropriate results returned back to the client. The run-time costs account for activities from the initiation of the operation to the receipt of the

appropriate result. The results shown are based on 200 performance measurements for each of the operations and relation sizes shown. The large sample measurement size was required to validate the results produced.

The results show that the project and select operation run-time costs grow in a linear fashion in relation to the size of the data being processed. The union and set difference operations run-time costs grow exponentially in relation to the size of the data being processed. The run-time cost is the mean of the 200 performance measurements. There was minimal deviation between the mean run-time cost and the performance measurements used in deriving the mean, usually with 90% of the performance measurements falling within ±10% of the mean. The deviation which did occur between measurements can be attributed to the limited clock granularity of the hardware involved, and to the underlying operating system.

Operation	Tuple Size	Run-time Cost	Tuple Size	Run-time Cost	Tuple Size	Run-time Cost	Tuple Size	Run-time Cost
Project	1	1.266	50	2.213	100	3.141	200	4.902
Select	1	1.179	50	1.436	100	1.701	200	2.218
Union	2	2.714	100	5.499	200	10.002	300	16.248
Set Diff	2	3.033	100	6.723	200	11.984	300	19.183

Figure 3. MRDB run-time costs

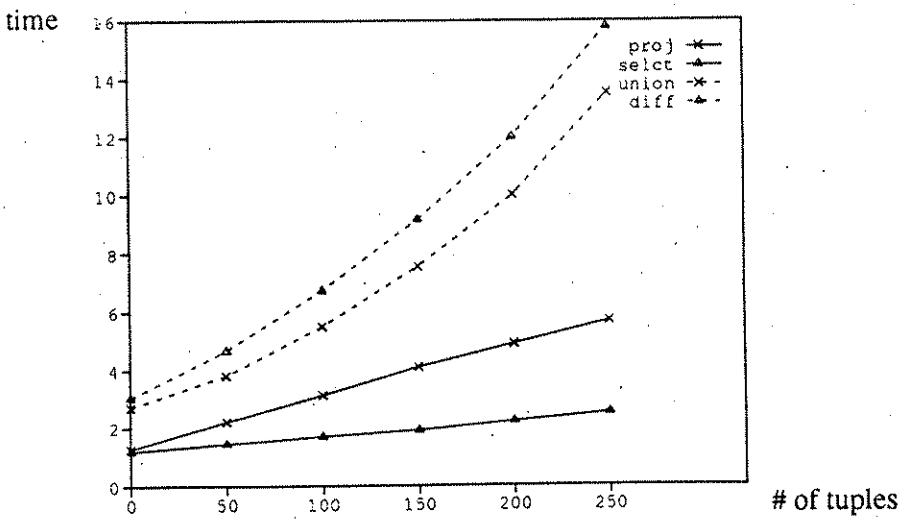


Figure 4. MRDB run-time costs

Our run-time cost results did show that run-time estimates could be derived not only based on the database operation being performed and relation size, but also on the number and types of attributes which make-up relations. However, the results also showed that such derived run-time estimates were heuristic in nature, and that no guarantee could be made that a given transaction's actual run-time cost would be as estimated. One of the primary contributing factors was the support of the underlying operating system. However, it is still possible to establish functions which generate acceptably accurate run-time estimates based on the physical characteristics of the data and the operations being executed, and that is what is implemented in the MRDB system.

The MRDB system maintains data on the physical characteristics of the relations in the database. When the scheduler is invoked it extracts the physical characteristics data for the relations being processed by a given transaction. This information is used in conjunction with the operation being performed to derive a run-time estimate. The run-time estimate is subsequently used in system scheduling decisions. An extract of system performance measurements conducted to verify that system generated run-time estimates closely approximated actual run-time costs is given in Figure 5. The solid lines show the system generated run-time estimate for the aggregate operations *avg* *max*, and *sum*. The dashed lines show the actual run-time costs for those operations based on 200 performance measurements. The system estimate closely approximates the actual cost.

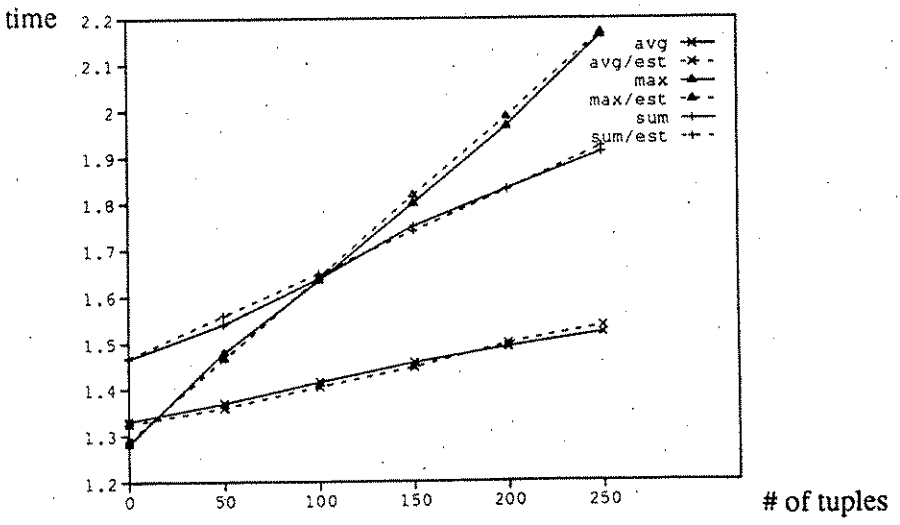


Figure 5. Run-time estimates versus actual cost

While no guarantee can be made for a given transaction, it is possible to state that a given percentage of transactions can complete within the run-time estimate generated by the system. Additionally, it can be stated that the run-time estimates generated will be within a given percentage of the actual run-time costs. For example, raising the system generated run-time estimates by 10% resulted in approximately 90% of the transactions accepted for processing by MRDB having actual run-time costs within the system generated values. The down-side of raising the estimate is that some transactions, whose actual run-time cost is below the system generated estimate, may be needlessly aborted. The percentage of these depends on the tightness of the temporal deadlines attached to the transactions.

5. Conclusions

A real-time database manager is one of the critical components of real-time systems, in which tasks are associated with deadlines and a significant portion of data is highly perishable in the sense that it has value to the system only if it is used quickly. To satisfy the timing requirements, transactions must be scheduled considering not only the consistency requirements but also their temporal constraints. In addition, the system should be predictable, such that the possibility of missing a deadline for a given transaction can be determined prior to the execution of that transaction or before that transaction's deadline expires.

In this paper, we have presented a relation database manager which possesses temporal functionality, developed for investigating real-time database issues. Since the characteristics of a real-time database manager are distinct from conventional database managers, there are different issues to be considered in developing a real-time database manager. For example, the use of run-time estimates in scheduling policies, and the ability to place temporal consistency constraints on database operations. MRDB was designed with the goal of providing an operational platform for conducting research on real-time database issues. Previous studies using simulated environments have provided valuable information with respect to real-time database issues. However, performance results in some of the simulated studies are sometimes contradictory since they make different assumptions about system environments. We feel that an operational environment for

investigating real-time database issues will eliminate some of the problems associated with simulated systems and provide valuable and applicable insights to real-time database issues.

The MRDB system is completely functional. The foundation now exists for studying real-time database issues in an operational environment. The results achieved in deriving and applying heuristic run-time estimates and the ability to attach temporal consistency specifications are promising. However, as with any active real-time database research project, there remains many technical issues associated with real-time database management which need further investigation. It is our goal to facilitate further development in this area. To that end we have oriented our work effort toward integrating and analyzing various conflict resolution mechanisms, to include optimistic concurrency control mechanisms based on the notion of dynamic adjustment of serialization order [Lin90]. We also plan to extend the system to a distributed database environment, capture system performance measurements, and improve the temporal functionality.

REFERENCES

[Abb88] R.Abbott, and H.Garcia-Molina, "Scheduling Real-time Transactions: a Performance Evaluation," Proceedings of the 14th VLDB Conference, Los Angeles, California 1988.

[Abb89] R.Abbott, and H.Garcia-Molina, "Scheduling Real-time Transactions with Disk Resident Data," Proceedings of the 15th VLDB Conference, 1989.

[Abb90] R.Abbott, and H.Garcia-Molina, "Scheduling I/O Requests with Deadlines: a Performance Evaluation," IEEE, Real-Time Systems Symposium, Orlando, Florida, December 1990.

[Bec90] R. C. Beckinger, "SDB User Manual," Technical Report, Dept. of Computer Science University of Virginia, Charlottesville, Virginia, September 1990.

[Ber87] P.A.Bernstein, V.Hadzilacos, and N.Goodman, "Concurrency Control and Recovery in Database Systems", Addison-Wesley Publishing Co., 1987.

[But90] M. Butterbrodt, and J. Green, "DOSE: A Vital Link in the Development of a Real-Time Relational Database Environment," Project Summary, Naval Ocean Systems Center, Jan. 1990.

[Car89] M.J.Carey, R.Jauhari, and M.Livny, "Priority in DBMS Resource Scheduling," Proceedings of the 15th VLDB Conference, Amsterdam, 1989.

[Hua90] J.Huang, J.A.Stankovic, K.Ramamritham, and D.Towsley, "On Using Priority Inheritance In Real-Time Databases," Dept. Computer and Information Science, University of Massachusetts, November, 1990.

[IEEE90] Seventh IEEE Workshop on Real-Time Operating Systems and Software, University of Virginia, Charlottesville, Virginia, May 1990.

[Lin89] K.Lin, "Consistency Issues in Real-Time Database Systems," Proc. 22nd, Hawaii International Conference on System Sciences, January 1989.

[Lin90] Y. Lin, and S. H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," 11th IEEE Real-Time Systems Symposium, Orlando Florida, December 1990.

[Liu88] Liu, J. W. S., K. J. Lin, and X. Song, "Scheduling Hard Real-Time Transactions," 5th IEEE Workshop on Real-Time OS and Software, May 1988, pp 112-260.

[ONR90] ONR Workshop on Foundations of Real-Time Computing, Washington, D.C., October 1990.

[Ozso90] G. Ozsoyoglu, "CASE-DB: A Real-Time Database Management System," Tech Rep., Case Western Reserve University, 1990.

[Sha88] L.Sha, R.Rajkumar, and J.Lehoczky, "Concurrency Control for Distributed Real-Time Databases," ACM SIGMOD Record 17, 1, March 1988, pp. 82 - 98.

[Sha91] L.Sha, R.Rajkumar, S. H. Son, and C. Chang, "A Real-Time Locking Protocol," IEEE Transactions on Computers, to appear.

[Sno87] R. Snodgrass, "The Temporal Query Language TQUEL," ACM Transactions on Database Systems, Vol. 12, NO. 2, June 1987, pp 247-298.

[Son88] S.H.Son, "Real-Time Database Systems: Issues and Approaches," ACM SIGMOD Record 17, 1, Special Issue on Real-Time Database Systems, March 1988.

[Son89] S.H.Son, and H. Kang, "Approaches to Design of Real-Time Database Systems," International Symposium on Database Systems for Advanced Applications, Seoul, Korea, April 1989, pp 274-281.

[Son90a] S.H.Son, and C. Chang, "Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment," 10th International Conference on Distributed Computing Systems, Paris, France, June 1990, pp 124-131.

[Son90b] S.H.Son, and J.Lee, "Scheduling Real-Time Transactions in Distributed Database Systems", 7th IEEE Workshop on Real-Time Operating Systems and Software, Charlottesville, Virginia, May 1990, pp. 39 - 43.

[Son90c] S.H.Son, "Real-Time Database Systems: A New Challenge," Data Engineering, Vol. 13, no. 4, Special Issue on Future Directions on Database Research, December 1990.

[Son91] S.H.Son, P. Wagel, and S. Park, "Real-Time Database Scheduling: Design, Implementation, and Performance Evaluation," The second International Symposium on Database Systems for Advanced Applications, Tokyo, Japan, April 1991, pp 146-155.

[Stan88] Stankovic, J., "Misconceptions about Real-Time Computing," IEEE Computer 21, 10, October 1988, pp. 10-19.