PARTIAL COMPUTATION IN REAL-TIME DATABASE SYSTEMS: A RESEARCH PLAN

Susan B. Davidson
and Insup Lee

MS-CIS-88-82
GRASP LAB 158

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

October 1988

Research Proposal Submitted to NSF.

Acknowledgements: This research was supported in part by NSF grants IRI86-10617, DCR 8501482, DMC 8512838, MCS 8219196-CER, U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027 and a grant from AT&T's Telecommunications Program at the University of Pennsylvania.

Partial Computation in Real-Time Database Systems: A Research Plan

Susan B. Davidson
Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

Research Proposal Submitted to NSF.

Abstract

State-of-the-art database management systems are inappropriate for real-time applications due to their lack of speed and predictability of response. To combat these problems, the scheduler needs to be able to take advantage of the vast quantity of semantic and timing information that is typically available in such systems. Furthermore, to improve predictability of reponse, the system should be capable of providing a partial, but correct, response in a timely manner. We therefore propose to develop a semantics for real-time database systems that incorporates temporal knowledge of data-objects, their validity, and computation using their values. This temporal knowledge should include not just historical information but future knowledge of when to expect values to appear. This semantics will be used to develop a notion of *approximate* or *partial computation*, and to develop schedulers appropriate for real-time transactions.

1 Introduction: What is a real-time database?

In *real-time* applications, programs must not only be *functionally* correct, but must meet *timing constraints*. For example, in tracking systems, input data must repeatedly be received from sensor processes. The data is then collectively examined and calculations are performed to determine if and where to move the sensor processes so as not to lose the object being tracked. In such a system, it is not enough that the calculations be performed correctly; the calculations must be performed quickly enough that the sensors can be alerted or moved in time to maintain their view of the object, or that defensive action can be taken. Input from the sensor processes must be received on a cyclic basis, the period of which is determined by the motion of the object being tracked. Furthermore, the time at which the sensor data was taken is important to the result of the calculation.

A critical component of these applications is the database, which is used to store external input such as environmental readings from sensors, as well as system information. Entries in the database are rarely deleted but constantly updated; hence sophisticated database management systems are necessary. However, state-of-the-art database management systems are typically not used in real-time applications due to two inadequacies: 1)*speed* and 2) *predictability of response* [1, 2].

The first inadequacy, speed, should be carefully considered. Database management systems are, and always have been, concerned with the throughput of transactions; much research has been done in this area (efficient buffering, indexing, design, query optimization, main memory databases, parallel architectures, etc.). While these techniques are important, they are non-specific to real-time databases. We wish to concentrate on the distinctive characteristics of real-time databases that can be used to improve the responsiveness of such systems.

One important distinction between real-time and non-real-time databases is how constrained the environment is. For example, to design a traditional relational database management system, one generally knows something about how attributes relate to one another (functional, multivalued dependencies, *etc.*), and something about the expected transactions and their relative frequency. One then designs the system to perform well for the expected transactions, yet have the relations in some appropriate normal form whenever possible. However, one still expects a large number of *ad hoc* transactions. In database applications for robotics or surveillance systems, most of the transactions are predefined, not only in the sense that the operations and data-items requested are known, but in that the time at which the results will be required is known. Furthermore, the time intervals during which data-items will be updated is predictable. For example, sensor processes may be known to take readings during some time interval of their periodic execution. The data may also have a validity interval: a citing can be used to extrapolate values within a small neighborhood

of the time it was made although it is only accurate for the real-time moment in which it was taken. The database will therefore know when to expect new values for the data representing the readings, and should be able to use this knowledge to schedule transactions using the new data to improve their response time. Furthermore, the completion time for these transactions may depend on the validity interval of the data computed; the transactions can be thought of as being triggered by the arrival of the data. In fact, this is what is frequently done in practice: designers of real-time database systems put not just data, but actual code into home-grown systems that allow them to take advantage of the detailed knowlege they have of their application. To design an *application independent* database management system that will be useful for real-time systems, we need to develop a *semantics* which will allow us to capture the temporal knowledge of data, the operations on the data, and times by which the operations will be expected to complete. We also need to consider schedulers that will take advantage of the future knowledge of transactions and their deadlines.

The second inadequacy is predictability of response: a computation must be *guaranteed* to complete by its deadline. This is not necessarily synonymous with "quickly". For example, payroll systems are "real-time" in the sense that checks for employees must be generated by the first of the month. The payroll system has a whole month to work on the problem, not just milliseconds. However, predictability of response is particularly difficult when the deadlines are short since the complex interactions of scheduling the CPU controlling the database, scheduling disk accesses, buffering, and concurrency control become important. While these factors need to be examined as a whole, we would like to improve predictability by guaranteeing a response by the transaction's deadline which is either a complete result or a *partial* result. This requires the computation to have an iterative or multi-phase nature. Furthermore, the computation should be *monotonic* in the sense that "goodness" of the answer is monotonically non-decreasing as the computation proceeds [3, 4]: Not only should the "precision" of the answer improve as computation proceeds, but an answer that is produced at one point in computation should never be contradicted by a later answer. An example of such a computation is the bisection method for finding the root of a function. The interval containing a root is initially very large, and keeps halving as the computation proceeds. At any point in the computation, the interval is valid; however, it is best defined when the endpoints of the interval converge to a single point, the root of the function. We would like to develop iterative techniques for querying databases that provide a partial answer at any point in computation, the goodness of which improves monotonically as computation proceeds.

As an example of how a partial answer to a database query could be useful, suppose that we have a distributed sytem of three blood bank databases. Each blood bank database maintains,

among other information, a relation of how many pints of each blood type is currently on hand. Suppose that type O- is dangerously low at hospital X, and X is trying to find out if there is any available within the network of blood banks to meet a current crisis. This query could be expressed as: "Is there a blood bank that has blood of type O-?", Thus, the query "Do you have blood of type O- ?" would be broadcast to each of the three databases, and the final answer would be the logical "or" of the responses from each of the databases. The initial partial anwer to the query would be "I don't know yet." This partial answer can be changed to "Yes" immediately some blood bank responds with a "Yes", regardless of whether all blood banks have responded. The answer becomes "No" only when all of the blood banks have responded negatively. However, at any point in time, there is some answer to the query that is correct. If hospital X then wanted to know "How much blood of type O- is there in the system?", the initial partial answer would be "At least 0", and would be improved by adding the total amount from each database as the information became available. For instance, if the first blood bank responded with "10 pints", the answer would be improved to "At least 10 pints." If the second and third blood banks responded with 5 and 25 pints respectively, the answer would become "Exactly 40 pints."

This example underscores several of the points that we have been making:

- It is an example of a real-time process in which the response must be predictable: Hospital X cannot wait indefinitely for an answer from the blood banks since in the worst case that there is no O- blood it must start rounding up donors to cover any anticipated crisis. However, its deadlines would be minutes (or hours, depending on how nervous hospital X is) rather than milliseconds.

- A "hand-coded" query system which anticipates this type of query probably would act as in the example, while a strict relational algebra system would not be optimized to give partial information.

- The "goodness" of the answer given to the user is monotonically non-decreasing with time. Given a partial answer "At least 10 pints." the user can infer that the total is *definitely not* less than 10 pints, and *possibly* any integer greater than or equal to 10 pints (11 or 1198, for example). Furthermore, the answer given at an earlier stage is never contradicted at a later stage.

In summary, we propose to develop general purpose databases management techniques for real-time database systems. To do this, we must address the primary problems of *speed* and *predictability of response*. To improve the predictability of response, we propose to look for methods of *incremental* computation of queries which can be used to generate *partial* answers in a timely

manner. To improve the thoughput of transactions in these systems where the queries are largely preknown, we propose to encorporate into the database as much temporal information as possible about when values are expected to be updated, and what the deadlines of transactions accessing these values will be. We will then develop schedulers appropriate for real-time transactions.

The rest of this proposal is organized as follows: In the next section we discuss partial computation of database queries, and present some preliminary ideas. We then briefly survey research in temporal databases, and discuss what additional features are needed to use these ideas in real-time databases. We also give insights into how the scheduling of transactions can be improved using this extensive temporal knowledge. Section 3 summarizes our expected contributions. The last two sections respectively contain a justification of the budget, and curriculum vitae of the principal investigators.

2 The Research Plan

2.1 Partial queries

Real-time systems define correctness as providing the correct result in a timely manner. If the timing requirements for a computation cannot be met, the computation fails. To relax this definition of correctness, one must either be willing to accept the results of computation late, or be willing to accept partial, poorer quality results in a timely manner. The first strategy interprets timing constraints as being "soft": the completion of a computation or set of computations has a value to the system which is expressed as a function of time. The system schedules these computations to maximize the total value to the system; however, it does not guarantee that all computations will be performed at their local maximum value [5]. The second strategy requires the computations to have an iterative or multi-phase nature. Furthermore, they should be *monotonic* in the sense that "goodness" of the answer is monotonically non-decreasing as the computation proceeds [3]. An example of this sort of computation was given in the introduction for a distributed blood bank database: The query "How many pints of O- blood are there?" could intially be answered by "At least 0", and be improved by adding the amount from each individual database as the values became available ("At least 15", "At least 20") until all databases had answered ("Exactly 40").

While "soft" timing constraints have recently been proposed for transaction management in real-time database systems [6], little work has been done on generating partial or "approximating" answers to queries. ([7] is a notable exception to this, and will be discussed later). Unless all structures necessary to the query are accessible, there is no notion of an answer. For example, in relational databases, a query $f(R_1, R_2, R_3, ..., R_n)$ can be thought of as some combination of the

tables R_1,..., R_n using relational algebra operators. For simple expressions involving one binary operator, the relationship of tables R_1, R_2 to the final result is not difficult to reason about: For example, if $f(R_1, R_2) = R_1 \cup R_2$, it is obvious that R_1 and R_2 each contain a part of the answer, although, in general, neither will contain the whole answer: R_1 and R_2 are both said to be *consistent approximations* of the final result. If $f(R_1, R_2) = R_1 \bowtie R_2$, then every tuple in the join is contained in both R_1 and R_2 , but each table may contain other tuples as well that do not participate in the join. Both are said to be *complete approximations* of the final result. Note in this case that a tuple of R_1 participating in the join with R_2 is a *partial description* of the tuple in the result since it may not contain all the fields in $R_1 \bowtie R_2$. However, for more complicated expressions like $f(R_1, R_2, R_3) = R_1 \bowtie (R_2 \cup R_3)$, it is difficult to reason about the information in R_2 and R_3 with respect to the final result.

The reason why conventional query languages (and the relational algebra in particular) do not seem to be amenable to an iterative method is that the relationship of individual tables (or whatever structure is used in the model) to the final result is not explicit. Using the semantic notions of complete and consistent approximations, we have recently presented a method of iteratively combining structures as they become available [8, 9]. That is, the user first specifies the semantic relationship of the answer to the query to the individual relations in the database. The system then combines the approximations as structures become available in such a way that a partial answer is always available. The partial answer consists of a complete approximation, from which the user can infer tuples that are definitely *not* in the answer to the query, and a consistent approximation, from which the user can infer tuples that definitely *are* part of the answer. If some of the structures are inaccessible due to concurrent, conflicting activity in the database or due to the fact that they are located at a remote node and take too much time to be shipped over the network, a partial answer can be constructed.

To motivate these ideas, we will walk through an example of a real-time monitoring system for a hospital and show how the query could be specified and answered iteratively. The system is intended to alert the hospital staff when a patient becomes critically ill ($CODE - RED$), and "immediately" provide them with complete and current statistics on the patient (such as blood and urine analysis or whatever other tests are being routinely done on the patient, and vital statistics that are being constantly monitored such as blood pressure and temperature). The database is widely distributed. The business office contains the usual patient information:

$$PATIENT(Name, Address, PatientCode, NextOfKin, TelephoneNum, ...).$$

The test lab contains results of tests performed on people:

$$LAB(PatientCode, TestDate, TestType, Results).$$

The intensive care unit contains a history of vital statistics on critically ill patients:

$$P - B(PatientCode, BedNum)$$

$$ICU(BedNum, Temperature, BloodPressure, Pulse, ...).$$

Each patient is connected to dedicated machines taking measurements, which detect critical conditions (such as a rapid rise in blood pressure). When a critical condition occurs, a flag is raised and the bed-number is sent to the central computer in intensive care. Note that at any given time, any number of patients occupying beds in the intensive care unit may have activated an alerter. This can be thought of as a series of relations ($ALERT_1$, $ALERT_2$,...) where $ALERT_i$ contains all bed-numbers that have activated an alerter since the time $ALERT_{i-1}$ was activated. A current report ($CODE - RED_i$) on the patients occupying the flagged beds must then be made available:

$$CODE - RED_i(PatientCode, Name, BedNum, NextOfKin,$$

$$(TestDate, TestType, Results)^*, (Temperature, BloodPressure, Pulse)^*)$$

This query will not necessarily generate a first normal form relation, since there are an unspecified number of test types and their results for each patient, as well as a partial history of vital statistics. The query may also wish to specify "the most recent" result of each test, or ask for a limited history of the results of each test to give the staff an overview of how the patient is reacting. Note that in this situation, the staff cannot wait an unbounded amount of time for the complete answer. Often a partial answer will give them enough information to determine what immediate action to take. This course of action can be improved as more information about the patient is obtained.

In addition to the above relations, suppose that we know the following:

1. Every person who goes code-red is registered as a patient in the business office: $PATIENT$ is a complete approximation of $CODE - RED_i$.

2. Every person who goes code-red is in the intensive care unit: ICU is a complete approximation of $CODE - RED_i$.

3. Every person who goes code-red has had some tests sent to the test lab: LAB is a complete approximation of $CODE - RED_i$.

4. Only people who activate the alerter go code-red: for each i, $ALERT_i$ is a consistent approximation of $CODE - RED_i$.

5. The relations are correct and complete.

6. The patient codes are unique.

Suppose that beds 1 and 2 simultaneously activate the alerter at time i ($ALERT_i = \{1,2\}$), and that very shortly later, bed 3 activates the alerter ($ALERT_{i+1} = \{3\}$). Intuitively, what we would want to do is augment the $ALERT$ relations with information from ICU, $PATIENT$ and LAB that pertains to the critically ill patients occupying the listed beds . This could be done at the time the alerter was activated by retrieving the local *PatientCode* from ICU, and sending the request off to the business office and lab. The remote nodes would then send the requested information, first about the patients in beds 1 and 2, and then for the patient in bed 3. However, if the remote nodes had been previously notified about who was in the intensive care units, they could have sent their complete approximations of the requested information as the information became available (*e.g.*, as tests were analyzed, they would forward information about anyone in the intensive care unit): ($ICU \bowtie LAB$). Thus, when the alerter was activated the necessary information could be locally extracted from the complete approximation.

In our system, either of these approaches could be taken. Given the semantic understanding of relations in the system relative to the abstract concept $CODE - RED_i$, we create a partial result, which monotonically improves with time. The partial result is represented by a *bounding pair* (A, B), where A is a complete approximation of the final result and B is a consistent approximation of the final result. For example, the bounding pair at the lab computer might be $(LAB \bowtie (\Pi_{PatientNum} ICU), \{\})$, the bounding pair at the business office $(PATIENT, \{\})$ and the bounding pairs at the intensive care unit $(\perp, ALERT_i)$, $(\perp, ALERT_{i+1})$. Given two bounding pairs for a query, (A_1, B_1), (A_2, B_2), we combine them into another bounding pair (A, B) where A is no "larger" a complete approximation than A_1 or A_2, and B is "at least as large" a consistent approximation as B_1 and B_2. That is, the new bounding pair is a better approximation of the final result since it squeezes the complete and consistent approximations closer together. This continues until there are no more bounding pairs to encorporate, or until A and B describe the same set of objects, *i.e.*, the answer is determined. For example, combining the bounding pair from the lab computer with the bounding pair from the business office would yield the bounding pair

$$((LAB \bowtie (\Pi_{PatientNum} ICU)) \bowtie PATIENT, \{\}).$$

Combining bounding pairs at the intensive care unit would yield

$$(\perp, ALERT_i \sqcup^{\flat} ALERT_{i+1}),$$

where $\sqcup^{\flat}$ can be thought of as the "union" operator. These could then be combined to yield the pair

$$((LAB \bowtie (\Pi_{PatientNum} ICU)) \bowtie PATIENT, ALERT_i \sqcup^{\flat} ALERT_{i+1}).$$

The final answer would be

$$\mathcal{P}((LAB \bowtie (\Pi_{PatientNum} ICU)) \bowtie PATIENT, ALERT_i \sqcup^{\flat} ALERT_{i+1}),$$

where $\mathcal{P}$ is a special operator that extends tuples in $ALERT_i \sqcup^{\flat} ALERT_{i+1}$ with the extra information from $LAB \bowtie (\Pi_{PatientNum} ICU)$ (it can loosely be thought of as the join of these two sets).

The system has several advantages:

1. It can be used to provide a partial, monotonically improving answer to a query. A partial result can be shown to be "correct" at any time in the sense that if an object is said to satisfy the query, it will never be shown *not* to satisfy the query as computation proceeds; if an object can be inferred to not satisfy the query in a partial result, it will never be shown to satisfy the query in an improvement to the answer.

2. It is not tied in to any data model in particular (although the example given was relational in flavor).

3. It detects anomalies in the database, which can arise either due to incorrect semantic understanding of the structures in the database, or due to errors contained in the database. For example, if the patient had been rushed to the intensive care unit in such a hurry that they were not admitted correctly and entered into the business office's database, an anomaly would arise when the patient activated the alerter: there would be an entry in the consistent approximation of $CODE-RED_i$ with no corresponding entry in its complete approximation in the business office.

A disadvantage of this approach is that the complete approximation of the query may be a very large set, and could take too long to enumerate as a partial answer. We would like to be able to use the method proposed in [7] to use rules as a shortened, but accurate, description of this set. For example, if the patient had had a routine series of tests, all of which came back with normal results, the system should avoid listing each test individually but abbreviate with "G-series normal". We would also like to understand the relationship of this approach to deductive databases.

2.2 Semantics of Temporal Objects

In the previous example, time was frequently associated with the data. For example, each patient repeatedly undergoes the same tests (blood and urine analysis, for example), the results of which should be indexed by the time at which the sample was taken. Patients in the ICU are also constantly monitored for vital signs (a history of information). Furthermore, data has "intervals of validity": samples for tests are generally collected every morning and analyzed early in the day. A test-result that is more than one day old is probably only of historical interest, but not of current relevance. $ALERT_i$ also has an interval of validity, based on the period on which vital statistics are being measured (or more gruesomely, on whether the patient is alive).

Although real-time systems that are actually being implemented are much more complex than this example, input is usually received from the outside world (via sensors), and the data received frequently has a time or interval of validity. Thus, any model that will be useful for a real-time system must have a semantics for temporal data objects.

2.2.1 Overview of Previous Research in Temporal Databases

There has recently been great interest in encorporating time in databases. A taxonomy of various efforts appears in [10], where they distinguish static databases, static rollback databases, historical databases, and temporal databases. *Static databases* are traditional databases which store a "current" snapshot of the real world. Updates overwrite the old information, and there is no way for users to explicitly access past snapshots. *Static rollback databases* extend static databases by storing old snapshots, indexed by time. An update transaction T maps the most recent snapshot S_1 to a new snapshot S_2. The time associated with S_2 is the time at which T occurred (*transaction time*). Updates can *only* be made to the most recent snapshot; the sense of "history" that is provided is the history of updates to the database, rather than the history of the real world. Thus, we can ask for the following type of historical information: "What did we believe to have been true at time t?".

In *historical databases*, a snapshot represents what the database currently believes to have been true at that time. Updates can be made to *any* snapshot, not just the most recent snapshot; old snapshots are not retained. The time associated with a snapshot is the time at which those values are currently believed to have been true (*valid time*). Thus, if at some point in time we ask "What do we currently believe to have been true at time t?" we may get a different answer than if we ask the same question at a later time. Furthermore, we cannot necessarily find out what we believed to have been true at some time in the past.

Temporal databases support both transaction time and valid time. Thus, we can not only ask what we believed to have been true at time t, but see how that belief *evolved* to what we currently believe to have been true at that time. The information retrieved is much more complex than a snapshot since it also contains information about transaction and valid time.

Other types of time also seem to appear in the literature, which is loosely categorized as "user-defined". For example, consider the process of promoting a faculty member [10]. To do this, a promotion letter must be signed validating the promotion. The letter is then forwarded to the bursar's office (or whatever office contains the database of faculty members and their ranks), and the data entered into the database. Valid time is the time at which the letter validating the promotion was signed; transaction time is the time at which the information was entered in the database. However, the effective date of the promotion may be retro-active or future to either the valid or transaction time, and is therefore said to be "user-defined".

2.2.2 Extensions for Real-Time Databases

Since databases in real-time systems are unlikely to be relational, we need a semantics of time that is not tied in with a particular data model. With few exceptions, however, the work that has been done to date in temporal databases has concentrated on consistingly extending the relational model, and the relational algebra [11, 12, 13] (exceptions to this are [14]). We would like to develop a semantics of temporal objects that is independent of a data model, by extending work currently in progress [15, 8, 9].

Our initial feeling is that *historical time*, with updates restricted to monotonically improving what we know to be true, will be sufficient for most real-time applications: Sensor readings are taken, but not corrected once they have been entered in the database. However, valid time must be interpreted to model what we know to be true abou the future, as well as the past. Future knowledge has not been widely used to date; notable exceptions to this are [16], who allows future updates, and [14], who mentions that one might want to model potential futures from a given present situation. The reason for this is that future knowlege is often imprecise. Projecting where a missile will be at some future time t can be represented by a range of possible positions, none of which are known to be true. As time progresses to t, this range will become smaller; when t is reached, the answer will be a single value. Note that this type of update is monotonic since the set of possible positions is always being reduced in size, and no new positions are being added. Future knowledge is also imprecise because the time at which a value will become true is not known precisely. For example, one may know that an object will be returning a certain position sometime in the period $[t_1, t_1 + P]$. However, the exact time at which that will occur is not known. Our

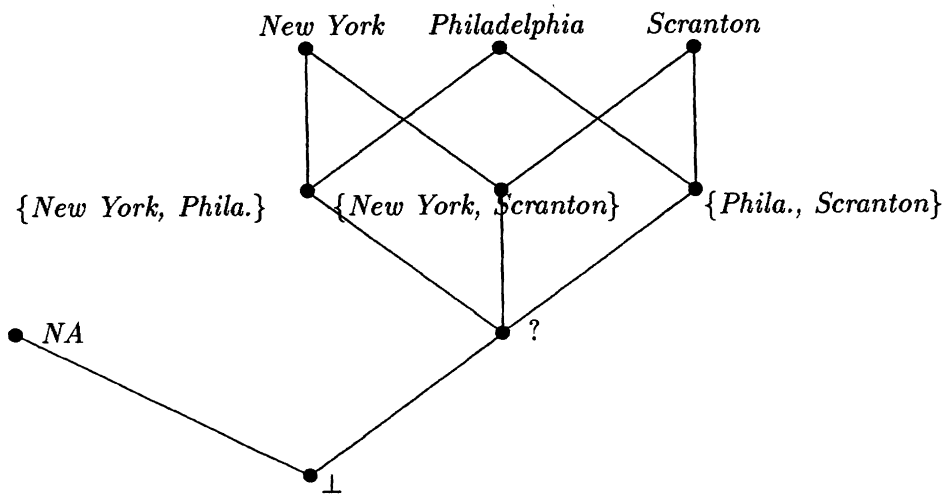

Figure 1: The extended domain of cities of the world.

model for historical data must therefore be able to accomodate information that is partial in the time domain as well as in the value domain.

Current work in temporal databases also does not seem to distinguish adequately between a value being *inapplicable* at time t ("NA"), and the value being *applicable but not available* at time t ("?") (this has been referred to as the "lifespan" of an object, but has not been completely developed [17, 11, 14, 18]). Note that NA and ? are both more informative than knowing nothing about the value of an object ($\perp$). Partial or incomplete information has been studied elsewhere, where the domain for an attribute is extended to include all possible subsets of the total values [19]. For example, if New York, Scranton and Philadelphia were the only cities in the world, the hierarchy of values for an attribute $CITY$ whose domain is the cities in the world would be as in figure 1. The benefit of using such a hierarchy is that information that is gained as time progresses can be used to improve a past approximation.

For example, suppose that we are interested in recording where people have lived, and represent this as a historical relation, $DOMICILES(Name, Year, City, State)$. We can think of each person as having an infinite number of tuples, from infinity past to infinity future; the times outside the lifetime of a person will yield inapplicable values for *City* and *State*. A partial relation containing information about Chris is in figure 2: Chris was born in 1985, and we have some information about where he lived in 1985 and 1986 but know nothing of his life since then. Suppose that we then find out that Chris lived in Pennsylvania in 1988, but we don't know if he lived in Scranton

Name	*Year*	*City*	*State*
Chris	[0,1984]	NA	NA
Chris	[1985,1985]	Philadelphia	Pennsylvania
Chris	[1986,1986]	New York	New York
Chris	[1987, ∞)	$\perp$	$\perp$

Figure 2: A partial DOMICILES relation.

Name	*Year*	*City*	*State*
Chris	[0,1984]	NA	NA
Chris	[1985,1985]	Philadelphia	Pennsylvania
Chris	[1986,1986]	New York	New York
Chris	[1987,1987]	?	?
Chris	[1988,1988]	{Philadelphia,Scranton}	Pennsylvania
Chris	[1989, ∞)	$\perp$	$\perp$

Figure 3: The partial DOMICILES relation after updating.

or Philadelphia. Not only can we insert some partial information about where Chris lived in 1988, but we can improve our historical information for 1987 since we now know that he was alive in 1987, *i.e.*, that information about *City* and *State* is applicable (see figure 3).

In the previous example, we used some sort of "temporal consistency constraint": if a person is known to be born at time t_1, and is known to be alive at time $t_2 > t_1$, then he is known to be alive any time between t_1 and t_2. The updates that we made to the relation were also "monotonic", in the sense that the information was only improved, but never retracted. The example also represents the type of temporal information and updating that occurs in real-time systems: in a robot application, sensory information may be received from distributed sensors and stored at a central database. This information may arrive out of order or even be lost (as in the partial information about where Chris lived in 1988). However, when values do arrive, they can be used to improve other values to which they are related that had previously been estimated (due to the value being lost, or delayed).

2.3 Schedulers for Real-Time Transactions

A key way in which real-time databases for surveillance or robotics applications differ from conventional databases is that many of the queries are predefined, not only in the sense that the data to be accessed can be anticipated, but that the deadlines are known. This knowledge can be used to improve the speed of the system in several ways: full or partial *pre-execution* of the queries, and

correct but non-serializable concurrency control mechanisms.

2.3.1 Pre-executing Frequently Executed Queries

Queries that are preknown or anticipated are commonly called *views*. To eliminate the overhead of computing a view every time it is accessed in a relational database system, several proposals have considered storing a *materialized view*. While storing the materialized view frequently simplifies queries on the view, updating the base relations incurs the additional expense of maintaining the materialized view. Due to this overhead, it was initially thought that materialized views should not be used to support real-time queries [20]. However, for several common types of view definitions there are *incremental* (or *differential*) techniques for updating the view; tuples can be selectively added to or deleted from the existing materialization rather than completely recalculating the view [21, 22, 23]. Recent performance comparisons indicate that these techniques perform well (in the sense of the "total cost per query" accrued to the system) if the view is very selective, queries on the view retrieve most of the view, and updates are infrequent [24].

However, the performance advantages of view materialization depend on the ability to use differential update techniques. Such techniques are not possible for *general* views whose definitions involve universal quantifiers, *i.e.*, the relational algebra "divide" operator which is not commutative, distributive, or associative. We have therefore been looking for other types of redundant data which can be used to improve the performance of queries on general views, and have recently proposed a new method called *semi-materilization* [25]. Semi-materialized views represent *partially executed* queries. They support efficient evaluation of queries on the view, but are easy to maintain (see [26] for a preliminary performance evaluation). We would like to continue to explore the usefulness of this and other techniques when there are several views defined over the same relations.

2.3.2 Non-serializable Execution

Serializability has been almost unanimously accepted as the appropriate correctness criteria in centralized and distributed database systems. Much research has been done in developing concurrency control techniques that guarantee serializability (*i.e.*, the concurrency control mechanism must guarantee than any legal schedule is equivalent to some serial execution of the transactions represented in that schedule). To extend this notion to real-time databases, one must ensure that schedules are not only serializable, but that the timing constraints of transactions can be met. Two efforts have recently appeared that extend locking [6] and timestamping [27] to real-time applications.

However, people building large real-time systems are unwilling to pay the price for serializability

[2]. Predictability of response is severely compromised due to waiting (in locking) or pre-emption (in timestamping). Furthermore, transactions may be serialized in such a way that the "most recent copy" of data is not used in a calculation, as in multi-version timestamping techniques, or locking techniques that force an update transaction to wait instead of making the most recent value available to the executing transaction. Real-time transactions frequently want the "current" view of the world, whether or not it is the result of a serializable execution sequence; or transactions may wish to see the world "as of" a certain time. Serializability is therefore not only too expensive in terms of predictability of response, but may not even represent the desired behavior. We would like to define an appropriate notion of correctness, and find ways to guarantee correctness while increasing the throughput of the system by using semantic knowledge of transactions, and the "as of" time of the world view they wish to use. We feel that ideas in [28, 29, 30, 31] will be applicable since data may already be replicated to speed queries (materialization or semi-materialization), and due to the presence of historical data (although data may perhaps stored at the same node rather than being distributed as in the above proposals).

Since many of the transactions in this environment are anticipated, it may also be possible to perform much of the scheduling of transactions in advance, borrowing ideas from scheduling periodic and aperiodic real-time jobs with precedence constraints in a multi-processor environment (see [32] to name just one and omit a multitude). Database transactions have *precedence constraints* among themselves based on the data-items that they read and write, which disallow certains interleavings of a set of transactions. Consider, for example, two transactions T_1 and T_2: T_1 reads the set of data-items $\{x, y\}$ and updates $\{y\}$; T_2 reads the set $\{x, y\}$ and updates (writes) $\{x\}$. The executions of these transactions cannot be interleaved in any way if a serializable schedule is to be achieved. However, if the scheduler allows transactions to be pre-empted (as in timestamping), it may be possible to avoid backing out the pre-empted transaction and instead use the partial computation when the transaction is resumed. Continuing with the previous example, suppose that x and y are large relations, and T_1 reads x to compute the total number of tuples in the relation. After reading x, suppose T_1 is pre-empted by T_2 who inserts 10 tuples and deletes 4 from relation x. T_1 does not have to re-read x to calculate the new total number of tuples, but can just add 10-4=6 to the previously calculated total to generate the appropriate update to y. We would like to look for generalizations of this technique, similar to incremental or differential updating performed on materialized views.

3 Expected Contribution

- **Definition of Issues Specific to Real-Time Database Systems.** While providing ef-

ficient access to data and improving the throughput of transactions has been the subject of database research for years, there has been almost no work on servicing transactions with timing requirements (exceptions to this are [6, 27]). The issues involved in building a general purpose (distributed) real-time database have not been clearly enumerated; this proposal is a starting point on which we will build.

- **Guaranteed Deadlines Using Partial Results.** A key way in which real-time database systems differ from traditional databases is the need for *predictable* response. The first way in which we propose to provide this is by *guaranteeing* an answer by a transaction's deadline (this was inspired by the notion of imprecise results for general computations proposed in [3]). While the answer may only be partial, it is guaranteed to montonically improve as time progresses. We have several promising results in this area, which we are generalizing to larger classes of queries and adapting to include rules [8, 9].

- **Semantics of Temporal Data Objects.** The notion of "partial computations" can also be used to predict future values: "Old" sensor data is never overwritten or discarded, and can be used to predict (partial) future values. Partial future values can be used to improve fault tolerance as well as predictability of response. Furthermore, new or out of sequence updates can monotonically improve past predictions using a notion of temporal consistency constraints.

- **Model of Real-Time Transactions and Notion of Correctness.** Another way in which we plan to provide predictable response is by capturing semantic and temporal information about transactions: their timing requirements, the data objects they access, how "recent" the values must be, and how "accurate" the values must be (in terms of partial results). A notion of "correct execution" for real-time transactions will then be developed, extending ideas in [28, 29, 30, 31].

- **Develop Efficient Schedulers.** The semantic information captured about transactions and the objects they access will then be used to develop efficient schedulers for real-time transactions. One idea is to preprocess as much computation as possible; here, we have some initial results on deciding what redundant data to store to optimize frequently executed queries [25, 26]. Another idea is to schedule actions of transactions in advance to eliminate unpredictable contention over shared objects [33]. A third idea is to use partial results to avoid completely recomputing a transaction if it is pre-empted.

This is a new and exciting area of research. Many of these preliminary ideas were drawn from

a workshop held last December [2] at which representatives from TRW, IBM and Boeing discussed the "real" real-time applications they were developing. *All* of these people pointed to the need for basic research in real-time distributed database systems. We also have access to a number of real-time applications in which to test the validity and usefulness of our ideas: a distributed multisensory robot system currently being developed here at the University of Pennsylvania; and a real-time distributed database for credit card calling being developed at AT&T. We feel that this environment will guide us in developing ideas that are not only theoretically sound, but practically important.

References

[1] H. Wedekind and G. Zoerntlein, "Prefetching in Realtime Database Applications," in *Proc. ACM-SIGMOD International Conference on Management of Data*, pp. 215–226, 1986.

[2] Presentations by TRW, IBM and Boeing. ONR Funding Workshop, San Jose, CA. December 1987.

[3] K. Lin, S. Natarajan, and J. Liu, "Imprecise Results: Utilizing Partial Computations in Real-Time Systems," in *Proceedings of the Real-Time Systems Symposium*, pp. 210–217, December 1987.

[4] D. W. Leinbaugh and M. Yamini, "Guaranteed Response Times in a Hard-Real-Time Environment," *IEEE Transactions on Software Engineering*, vol. 6, pp. 1139–1144, December 1986.

[5] E. Jenson, C. Locke, and H. Tokuda, "A Time-Driven Scheduler for Real-Time Operating Systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 112–122, December 1986.

[6] R. Abbott and H. Garcia-Molina, "What is a Real-Time Database System?," in *4'th Workshop on Real-Time Software and Operating Systems*, pp. 134–138, July 1987.

[7] T. Imielinski, "Intelligent Query Answering in Rule Based Systems," *The Journal of Logic Programming*, vol. 4, pp. 229–257, September 1987.

[8] P. Buneman, S. Davidson, and A. Watters, "Querying Independent Databases," *Information Sciences: An International Journal*, 1988. To appear.

[9] P. Buneman, S. Davidson, and A. Watters, "A Semantics for Complex Objects and Approximate Queries: Extended Abstract," in *Proceedings of the 6'th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, March 1988.

[10] R. Snodgrass and I. Ahn, "A Taxonomy of Time in Databases," in *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pp. 236–246, 1985.

[11] J. Clifford and A. Tansel, "On An Algebra for Historical Relational Databases: Two Views," in *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pp. 247–265, 1985.

[12] G. Ariav, "A Temporally Oriented Data Model," *ACM Transactions on Database Systems*, vol. 1, pp. 499–527, December 1986.

[13] R. Snodgrass, "The Temporal Query Language TQuel," *ACM Transactions on Database Systems*, vol. 12, pp. 247–298, June 1987.

[14] J. Clifford and D. Warren, "Formal Semantics for Time in Databases," *ACM Transactions on Database Systems*, vol. 8, pp. 214–254, June 1983.

[15] P. Buneman and A. Ohori, "Using Powerdomains to Generalize Relational Databases," *Theoretical Computer Science*, 1988. To appear.

[16] S. Gadia and J. Vaishnav, "A Query Language for a Homogeneous Temporal Database," in *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pp. 51–56, 1985.

[17] J. Clifford and A. Croker, "The Historical Relational Data Model (HDRM)," in *Proceedings of the International Conference on Data Engineering*, pp. 528–537, February 1987.

[18] A. Segev and A. Shoshani, "Logical Modeling of Temporal Data," in *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pp. 454–466, 1987.

[19] T. Imielinski and W. Lipski, "Incomplete Information in Relational Databases," *Journal of the ACM*, October 1984.

[20] G. Gardarin, E. Simon, and L. Verlaine, "Querying Real-Time Relational Data Bases," in *Proceedings of the IEEE-ICC International Conference*, pp. 757–761, May 1984.

[21] S. Koenig and R. Paige, "A Transformational Framework for the Automatic Control of Virtual Data," in *Proceedings of the Seventh International Conference on Very Large Data Bases, Cannes, France*, September 1981.

[22] O. Shmueli and A. Itai, "Maintenance of Views," in *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pp. 240–255, June 1984.

[23] J. Blakeley, P. Larson, and F. Tompa, "Efficiently Updating Materialized Views," in *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pp. 61–71, May 1986.

[24] E. Hanson, "A Performance Analysis of View Materialization Strategies," in *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pp. 440–453, May 1987.

[25] M. Kamel, S. Davidson, and E. Clemons, "Semi-Materilization: A Technique for Optimizing Frequently Executed General Queries," August 1987. Dept. of Computer and Information Science Tech. Rep. MS-CIS-87-66, University of Pennsylvania, Philadelphia, PA 19104.

[26] M. Kamel and S. Davidson, "Semi-Materialization: A Performance Analysis," November 1987. Dept. of Computer and Information Science Tech. Rep. MS-CIS-87-100, University of Pennsylvania, Philadelphia, PA 19104.

[27] S. Son, "Using Replication for High Performance Database Support in Distributed Real-Time Systems," in *Proceedings of the Real-Time Systems Symposium*, pp. 79–86, December 1987.

[28] H. Garcia-Molina, "Using Semantic Knowledge for Transaction Processing in a Distributed Database," *ACM Transactions on Database Systems*, vol. 8, pp. 186–213, June 1983.

[29] B. Kogan and H. Garcia-Molina, "Achieving High Availability in Distributed Databases," in *Proc. 3'rd International Conf. on Data Engineering*, February 1986.

[30] B. Kogan and H. Garcia-Molina, "Update Propagation in Bakunian Data Networks," in *Proceedings of the 6'th Conference on Principles of Distributed Computing*, August 1987.

[31] D. Skeen and D. Wright, "Increasing Availability in Partitioned Networks," in *Proceedings of the 3'rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 290–299, April 1984.

[32] S. Cheng, J. Stankovic, and K. Ramamritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems," in *Proceedings of the Real-Time Systems Symposium*, pp. 166–174, December 1986.

[33] J. Stankovic and K. Ramamritham, "The Design of the Spring Kernel," in *Proceedings of the Real-Time Systems Symposium*, pp. 146–157, December 1987.