REAL-TIME EXTENSIONS TO A RELATIONAL DATABASE

by

Anthony P. DiPesa Jr.

Submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING
AND COMPUTER SCIENCE

in partial fulfillment of the requirements

FOR THE DEGREES OF

BACHELOR OF SCIENCE

and

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1987

Signature of Author_____
Department of Electrical Engineering and Computer Science, May, 1987

Certified by_____
Prof. Nancy Lynch, Thesis Supervisor

Certified by_____
Dr. Craig Thompson, Company Supervisor

Accepted by_____
Dr. Arthur Clarke Smith, Chairman
Departmental Committee on Graduate Students

REAL-TIME EXTENSIONS TO A RELATIONAL DATABASE

by

Anthony P. DiPesa Jr.

Submitted to the Department of Electrical Engineering and Computer Science on May 15, 1987, in partial fulfillment for the degrees of Master of Science and Bachelor of Science in Computer Science.

Abstract

Today's real-time applications manage rapidly changing data in an "ad hoc" manner. This approach is taken because existing database management systems are not designed to handle real-time data. Several techniques are described which would allow a conventional database system to handle real-time data. These techniques include query optimization, minimizing the number of database queries that must be made, allowing processes to share memory with the database, prioritizing updates, and setting an update sampling frequency. The implementation of these techniques, in the form of general-purpose triggers, real-time triggers, real-time fields, and a view cache with incremental updates, is described. Their effectiveness is determined using timing analyses and computational analyses. The goal of this thesis is to extend a relational database to enable it to maintain the data of a real-time application.

Thesis Supervisor: Nancy Lynch
Title: Professor, Computer Science

Company Supervisor: Dr. Craig Thompson
Title: Senior Member of Technical Staff, Texas Instruments

Acknowledgements

I would like to thank Dr. Craig Thompson for all the feedback and encouragement he has given me throughout this research. Thanks to Steve Corey, Steve Scott, and the numerous other members of the Computer Science Center at Texas Instruments for their help. I also thank Prof. Nancy Lynch and Dr. Paris Kanellakis for their suggestions during the past several weeks.

Thanks to the brothers of the MIT Chapter of Beta Theta Pi fraternity. They have been my closest friends for the past five years. Finally, my deepest thanks to my brothers and sisters, Robert, David, Elaine, and Christine, and my mother and father, Carol Anne and Anthony DiPesa. Their love and their confidence in me have been my source of strength here at MIT.

Don't give up. You know it's never been easy. Don't give up. 'Cause I believe there's a place, there's a place where we belong.

- Peter Gabriel -

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Problem

Existing databases have been designed to maintain fairly static data and provide an interface through which this data can be modified, operated on, and retrieved. These existing databases are adequate for use in applications which do not maintain real-time data.

However, there is a significant number of applications in which it is essential to be able to manage rapidly changing data as well as more static data. Current relational database management systems were not designed to manage this real-time data. They do not handle base table updates or query computation in a fast enough or smart enough manner to achieve real-time behavior in the database system.

As a result, real-time applications are prevented from storing real-time data in relational database. They must maintain this data separately, outside the database. Thus, a great deal of control over the data is lost. In addition, the database interface capabilities and the power of the database operators are not made available for use with this real-time data.

A question which must be answered is, "Can existing relational database managers be modified to allow for the handling of real-time data?" If they can, then real-time applications can use these modified database systems to maintain all of their data,

including real-time data. If not, then new relational database management systems must be designed specifically to handle this real-time data.

1.2 The Approach

This thesis discusses a set of techniques which might enable real-time behavior to be achieved in a database system. These techniques include query optimization, minimizing the number of database queries that must be made, allowing processes to share memory with the database, prioritizing updates, and setting an update sampling frequency. Several mechanisms are identified which make use of these ideas. They are general-purpose triggers, real-time triggers, real-time fields, and a view cache with incremental updates.

An implementation of these mechanisms in an existing relational database system is described and an analysis of each mechanism is given. The ability of the extended database system to handle real-time data is compared with that of the original system. This analysis shows that an existing relational database management system can be extended to allow for real-time applications to maintain all of their data, static and rapidly changing, in the database.

1.3 Definition of Real-time

Many ideas of real-time exist. In some cases, real-time is used to signify "fast" or "faster". A system which processes data quickly is often considered to be a real-time system. An alternative idea of a real-time system is one which yields a response after a "small" number of processing cycles have passed. Given more processing cycles to work with, the system refines its response.

A more intuitive concept of real-time is discussed in [9] and [10]. A system is said to have a real-time response if its response time is of the same order as the time scale in which environmental events occur. This definition begins to suggest a notion

of real-time which concentrates on the setting or environment in which the system is operating.

1.4 The Need for Real-time Database Systems

The need for real-time database systems is application driven. An important application area which highlights this need is the real-time monitoring of dynamic objects. In many cases, dynamic objects can be monitored by a real-time system which does not make use of a database.

However, a database provides many useful services which a real-time system would require. These services include controlling the access to data items by users and by programs or processes, thereby providing a degree of security, consistency, and reliability which is usually desirable. The database serves as an interface between programs or processes by allowing them to share data items, decoupling them from the data, and providing an effective method of communication between them. It also provides users with an interface through which data can be updated and queried upon.

1.5 Applications of a Real-time Database System

A detailed description of several applications of a real-time database system for monitoring dynamic objects is now given. These applications, are from manufacturing, business, and the military. The diversity of these application areas indicates the wide range of use for a real-time database system.

1.5.1 Process Control

The first application discussed is process control. In an industrial automation or process control environment, there may be several robots operating in a factory. These robots may be assembling cars, painting parts, welding pieces of metal, or performing other tasks. While performing their tasks, the robots are moving in real-time. The

problem is to accurately monitor the position of each robot in the factory and to monitor the production of the item or items on which the robots are working. In addition, it is often desirable to display the positions of the robots in some manner and to maintain a display of the production level.

If a database is used to store the static information about the factory, such as the data on the workers in the factory, static data on the robots in the factory, and data on the parts used in production, then it should also be used to maintain the dynamic information dealing specifically with the production process and the movements of the robots. As mentioned above, the control, security, and interfacing capabilities provided by the database are significant considerations. In addition, the database will then serve as a central repository for all the information relating to the factory, not only the static information.

One could design the database so that all information, static or dynamic, about the robots is kept in one database table. The robot position is a real-time data value since the robots are capable of moving in real-time. Thus, it must be possible to update the database in real-time to reflect the changes in the values of the positions of the robots.

There is a set of sensors which is responsible for tracking the position of each robot and sending this updated information to the database. Concurrently, a process displays the current position of each robot in some useful fashion. There is a second set of sensors monitoring the production level of the item(s) being produced. Again, a display process is responsible for presenting a real-time display of the production process. For example, a real-time bar chart shows the amount of material being produced.

The robot sensors can be thought of as one process which is responsible for updating database values. They are updating the value of the real-time data field, the position field, of the robot table. The display can be thought of as a second process which is responsible for displaying the current values of the real-time attribute, robot position. In order for the current robot position values to be displayed properly, they must first be retrieved from the database. Both updates to the database and retrievals

from the database must be done in real-time if real-time behavior is to be achieved.

1.5.2 Business Application

In a hypothetical business application, information on several companies is stored in an ANALYST database. This information consists of static data such as quarterly sales, revenues, and profit figures, as well as volatile data such as the current price of each company's stock. One process monitors the New York Stock Exchange and updates the value of the price of each stock in the database. A second process displays, in real-time, a bar chart of the current prices of the stocks of all high yield companies. In order to accurately display the current value of these stock prices, the process issues a query to select out stocks whose earnings per share is greater than $5. Updates and queries to the ANALYST database must both be real-time.

1.5.3 Military Application

In a military application, there is information on vehicles such as planes, ships, and tanks stored in a database. Much of the information, such as model, manufacturer, cost, capacity, crew, and home base, is fairly static for these vehicles. The current location of a vehicle is a real-time data value. Again, one process updates the database with the new location values while a second process displays the locations of the particular vehicles in which the user is interested. Updates and queries to this military database must be performed in real-time in order to maintain this real-time display.

In a planning/replanning system, plans are generated based on the positions of these vehicles. Therefore, database accesses must be real-time if the generated plans are to be consistent with the current state of the world.

1.6 Organization of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 is a discussion of the techniques identified for achieving real-time behavior in a database. Chapter

3 describes the implementation of these techniques in RTMS, the relational database management system on the Texas Instruments Explorer Lisp Machine. Chapter 4 is an analysis of each of these real-time techniques. They are analyzed on the basis of their effectiveness in allowing real-time applications to be built on top of RTMS. Chapter 5, the final chapter, discusses the implications of this research and suggests directions in which it can be extended in the future.

Chapter 2

Techniques for Achieving Real-Time Behavior in a Database

2.1 Introduction

There are several techniques that can be considered for achieving real-time behavior in any database system. These techniques focus on database updates and on query computation. Real-time behavior can be achieved by making database updates and query computation faster, or by making the database smarter. Both of these issues are addressed in this chapter.

Although special-purpose hardware support may be useful for making certain database operations faster, it does not increase the functionality of the database system. It is also an expensive solution. Therefore, hardware solutions are not considered in this thesis. All of the real-time methods considered here are software-based. They include query optimization, minimizing the number of database queries that must be made, allowing processes to share memory with the database, prioritizing updates, and setting an update sampling frequency.

2.2 Query Optimization

The first technique considered, and one which has been discussed often in the literature, is query optimization. The goal of traditional query optimization, as discussed

in [12], is to produce a set of algebraic expressions which are equivalent to the given query expression, determine which expression has the lowest evaluation cost, and evaluate it in place of the original query. In [6] and [7], Johnathan King discusses semantic query optimization, which is based on a knowledge of the semantics of the domain and a knowledge of the structure of the database. Queries are transformed into semantically equivalent ones which require less computation to be evaluated.

A problem common to real-time systems has led to the introduction of a new type of query optimization in this thesis. In a real-time database system, most notably one in which dynamic objects are being monitored, it is very likely that the same or similar queries will be made repetitively. For example, consider a real-time system which is displaying, every two seconds, the positions of a group of planes which satisfy some selection criteria W.

A query of the form,

```
(RETRIEVE 'PLANES 'WHERE W 'PROJECT '(PLANE-ID POSITION))
```

will have to be made to the database before each display, in order to get the current position value for each plane satisfying the predicate W. If this same query will be made repeatedly, it is desirable to decrease, or minimize if possible, the amount of computation required to produce the query result each time.

Therefore, a new concept is introduced, query optimization which takes into account updates to the base table(s) from which the query result is computed. The idea is to reduce the amount of computation that must be performed in order to recalculate the value of a query after updates have been made to the base table(s) on which that query depends.

In [3], an algorithm is proposed for differentially updating views after updates to base tables on which the views depend. A relational database consists of both base relations and derived relations, or views. A view is defined by a relational expression over a base relation or relations. The view can either be a virtual view or a materialized (stored) view.

The processing of a query can be done faster if a view is kept materialized, that is, if the value of the view is stored, so that it can simply be retrieved without accesses to the base relation(s) on which that view depends. However, if updates are being made to the base relations, it is necessary to perform some computation in order to maintain the materialized view. [3] argues that complete re-evaluation of the view from the base relations is wasteful and the cost involved is often unacceptable.

In a real-time system, especially one for an application in which real-time dynamic objects are being monitored, the same query will likely be made repeatedly and updates to the base tables will be performed often. The database view which represents the current value of this query must be maintained with respect to these updates. Thus, the computation needed to maintain the view must be reduced. [3] suggests that this computation can be reduced significantly by examining the updates to the base relation(s) and determining which are relevant and which are irrelevant to the view. Then tuples are added to, and/or deleted from, the previous value of the view based on the relevant updates made to the base relations since the last computation of the view.

A simple example of this is a select view. If a selection is performed on a relation R1 based on a where-clause W1, the tuples representing the value of this selection may be stored in a select view S1. Then an update is made to the base relation R1. This update may consist of a set of inserted tuples I1 or a set of deleted tuples D1. In order to maintain the materialized view S1, the new view value must then be computed.

After an insertion, the new view value can be computed differentially by inserting into S1 the relevant tuples, those tuples which satisfy W1, in the set I1. After a deletion, the new view value can be computed differentially by deleting from the result the relevant tuples in D1. According to the paper, the computation required to perform this procedure after an insert should always be less than that required to completely re-evaluate the view from the base relation. After a deletion, the computation required to perform this procedure should be less than that required to completely re-evaluate the view as long as the cardinality of D1 is much smaller than the cardinality of the new relation R1', where R1' = R1 - relevant deletions in D1. The proposed algorithms for differentially re-evaluating project and join views are treated similarly in [3].

2.3 Minimizing Database Queries

A second technique, which is examined in this thesis, of achieving real-time behavior in a database is that of minimizing the number of queries that a user or process must make to the database. Each query that a user or process makes requires a certain amount of time and database computation in order to determine its value. By minimizing the number of queries that are made to the database, we will minimize the amount of computation which the database system must perform. This will free the computational resources of the system so that those queries which must necessarily be made can be processed more rapidly.

When there are multiple users or processes operating on the same database, a given user or process is likely to be unaware of all the updates being made to the database. In each of the three real-time database systems described previously (the process control system, the business application system, and the military system) there was one process updating the database and a second process displaying some aspect of the current database state. The display process in each system is unaware of the updates that are being made to the database. Without any knowledge of the database updates that are being made by the first process, the second process must query (poll) the database for the current value of the particular view it is displaying before every redisplay.

However, if the display process were given some knowledge about database updates, some of these queries might be eliminated. If it were alerted about each database update which affects the view it is displaying, the number of queries it must make would be reduced. In addition, if the display process received no alerts about updates, then it could correctly assume that the view value has not changed since the last display refresh.

A group at Texas Instruments Johnson City is implementing triggers in the Pathfinder database system precisely for this purpose [4]. Pathfinder will be a Unix-based real-time database system for process control. It will be used in conjunction with sensors, controllers and displays. When the system is employed in a real-time process

control environment, these triggers will reduce the number of queries that must be made to the database.

When updates are made to a relation from which real-time data is being displayed, the new data values will be sent directly to the display. Thus, a query will not have to be made before each display operation. Only the updated tuples which affect the view being displayed are needed by the display process, and the triggers ensure that these tuples are properly sent to the display process. With this added functionality, the database becomes an active database rather than merely being a passive repository.

Another related use of triggers has been previously explored in the design of System R [2]. In the design of this system, triggers are used to maintain constraints between relations when updates are being made to the relations. The user would be allowed to define triggers on all types of updates to a relation to enforce these constraints.

For example, a COMPANY database might have a relation which contains a tuple for each employee in the company. The tuple has attributes including EMPLOYEE-NAME, SALARY, and DEPARTMENT. The database also has a relation for each department. Tuples in this relation have attributes including DEPARTMENT-NAME and NUMBER-OF-EMPLOYEES. The user would be able to define a trigger on an insert into the EMPLOYEE relation which would increment by one the value of the NUMBER-OF-EMPLOYEES attribute in the tuple in the DEPARTMENT relation corresponding to the new employee's department. Triggers for deletes and modifies to the employee relation would be defined analogously.

2.4 Shared Database Memory

A third real-time technique considered here is to allow processes to share memory with a database. Traditionally, databases work to prevent direct access to their data. Data accesses are only allowed through the interface.

This control is good in many instances. It prevents the storage of invalid data in the database and prevents unauthorized data accesses. However, this control can

slow down data accesses too much to allow for operation in a real-time environment. If processes are allowed to share memory with the database, they can access it very quickly. With this approach, however, a certain amount of control over the data is lost and concurrency control is relaxed.

2.5 Prioritized Updates

A fourth technique for achieving a real-time response in a database system is often necessary when the volume of updates is extremely high. The idea is to prioritize the updates and perform high priority updates first, while queueing lower priority updates to be done after the high priority updates have been completed. This technique addresses another definition of real-time given in [9]. This definition of a real-time system is one which guarantees a response after a fixed time has elapsed, where that fixed time is given as a part of the problem statement.

The particular application determines the maximum time by which a response must be given in response to certain queries. Then updates to base tables are prioritized based on the values of the response times specified. High priority updates are performed first and the views affected by these updates are recomputed so that their current values are available by the maximum time specified. Once these views have been computed, lower priority updates are performed.

2.6 Update Sampling Frequency

A more straightforward method of ensuring that there is enough computation time available to recompute certain views is to set a sampling frequency for updates. In a real-time system for industrial automation, one may find sensors picking up updated data values faster than the database can process these values. Since the database can only process updates and update its views in a given time, a maximum frequency for the sampling of updates from the sensors can be set. If this maximum frequency is set properly, there will be enough computation time left to recompute the needed views

before the display process must have those new view values. The disadvantage of this approach is that unsampled updates are not reflected in the database.

Chapter 3

Implementation of Real-time Methods in RTMS

3.1 Introduction

This chapter describes the mechanisms that have been implemented in RTMS [1] in order to increase the functionality of the system, improve the system's response, and enable it to support real-time applications such as those previously described. These mechanisms include a general-purpose trigger mechanism, real-time triggers, a view cache with incremental updates, and a mechanism which allows pointers to be set directly into the data values in an RTMS database.

Triggers address the problem of minimizing the number of queries that the user must ask of the database. The view cache with incremental updates addresses the problem of decreasing the amount of computation that must be performed in order to respond to a user-defined query. It also indirectly addresses the idea of prioritizing updates. Direct pointers address the problem of decreasing the time required to perform base table updates and read data values from the database.

Implementation details of the various mechanisms are given in this chapter. In the next chapter, the effect of implementation choices on system flexibility and performance are discussed. Since RTMS and its existing structure have largely influenced the implementation of all of these mechanisms, a necessary discussion of RTMS is given

first.

3.2 RTMS

RTMS, Relational Table Management System, is a relational database manager on the Texas Instruments Explorer Lisp Machine. RTMS was designed and implemented "to realize the objective of long term management of data in a Lisp environment" [11]. The close integration of RTMS, a graphics window system, and a menu-based natural language interface makes the system particularly useful in many application areas.

RTMS implements most of the standard relational database capabilities. Within a database, relations, views, attributes, and secondary indices may be defined, modified, and destroyed. INSERT, DELETE, and MODIFY routines exist to allow for the update of tuples in a relation. The relational algebra operators RETRIEVE, JOIN, UNION, DIFFERENCE, and INTERSECTION are also supported.

However, RTMS currently lacks some features found in many relational database systems. There is no secure protection system for the data in an RTMS database, and no locking scheme to prevent concurrent access of data by multiple users or processes. In addition, RTMS only allows for one database to be accessed at a given time, and the tables of that database must be resident in virtual memory before they can be operated on.

The most notable feature of RTMS is its single language environment. All relational operators are implemented with lisp functions. Therefore, they can easily be embedded in lisp programs. This feature allows for the passing of parameters between database operators and other lisp functions without any specialized mechanism. RTMS calls have the same syntax as lisp function calls, with database, relation, and attribute names following the same rules as names of lisp variables. Also, any lisp object may be stored in an RTMS relation.

In addition to this flexibility, RTMS allows great flexibility in the definition of tuple implementations and index structures. Currently, list, flavor, and structure tuple

implementations, and heap, hash, and avl-tree index structures are supported. However, a user is allowed to define other data structures provided he supplies RTMS with an appropriate set of functions with which these new data structures may be properly accessed. It is this flexibility which makes RTMS a desirable database management system to work with.

3.3 Real-time Mechanisms in RTMS

3.3.1 General-Purpose Triggers

The first real-time mechanism implemented in Extended RTMS addresses the problem of minimizing the number of queries that a user or process must make to the database. Previous work has focused on special-purpose solutions to this problem.

The signalling mechanism being developed in the Pathfinder system discussed above will send a message to the display process when updates are made to a table from which real-time data is being displayed. With the trigger mechanism described in the System R design, the user is able to set constraints between relations in a database, so that an update to one relation causes a corresponding update to a second relation in the database. Neither of these special-purpose mechanisms is flexible enough to allow for the full range of actions that a user might want to be taken after an update to a base table in the database.

In Extended RTMS, a general-purpose trigger mechanism has been implemented to address this problem. This mechanism focuses on updates to any base relation in the database and is flexible enough to allow for all the possible implications that such an update might have in the database system. The trigger mechanism exhibits characteristics of both the trigger mechanism being developed in the Pathfinder system and the trigger mechanism described in the System R design. This section shows that it has the functionality of both of these systems and more.

Triggers can be defined in RTMS by both the user and the system itself. When a trigger is defined by either the system or by an RTMS user, it is stored as a tuple in the

SYSTEM-TRIGGER system relation. A trigger is a tuple consisting of the following attributes:

```
(NAME, RELATION, TYPE, WHERE-CLAUSE, ACTION,
 PRIORITY, ACTIVEP, APPLICATION-FIELD, DOCUMENTATION)
```

RELATION is the base relation on which the trigger is defined. TYPE is the type of update, insert, delete, or modify, on which the trigger is defined. WHERE-CLAUSE is an arbitrary lisp predicate which is tested on the list of updated tuples. ACTION is an arbitrary lisp form which is evaluated when evaluation of WHERE-CLAUSE on the list of updated tuples returns a non-null result. PRIORITY is a number, 0 or 1, which is used to determine which triggers should be executed first. PRIORITY is 0 if the trigger is system defined and 1 if the trigger is user-defined. Triggers with a priority of 0 are executed before triggers with a priority of 1. ACTIVEP is T when the trigger is currently active. Only active triggers are executed. APPLICATION-FIELD is a field in which the user may store any application specific information needed by the trigger.

Each time an update is made to a base relation, all the triggers that are defined on that relation, are of that update type, and are active, are retrieved from the SYSTEM-TRIGGER system relation. For each trigger, the following is done: the trigger where-clause is evaluated on the list of updated tuples. The result of this evaluation is a list of copies of the updated tuples which satisfy the where-clause. If this list is not null, then the action-clause of the trigger is evaluated.

If the update is an insert, then a list consisting of a copy of each inserted tuple which satisfies the trigger where-clause is stored in the global variable **triggered-inserted-tuples**. The action clause of the insert trigger may reference the list **triggered-inserted-tuples**. If the update is a delete, then a list of copies of deleted tuples which satisfy the trigger where-clause is stored in the global variable **triggered-deleted-tuples**. The action clause of the delete trigger may reference the list **triggered-deleted-tuples**.

If the update is a modify, then the modify trigger where-clause is evaluated on the list of modified tuples before the modify operation and on the list of modified tuples after

the modify operation. A list of copies of pre-modification tuples which satisfy the modify trigger where-clause is stored in the variable **triggered-deleted-tuples** and a list of copies of post-modification tuples which satisfy the modify trigger where-clause is stored in the variable **triggered-inserted-tuples**. The action-clause of the modify trigger may reference the list **triggered-deleted-tuples** and/or the list **triggered-inserted-tuples** Thus, different actions may be taken for the pre-modification tuples than for the post-modification tuples.

Multiple triggers are handled serially, in the order in which they are retrieved from the SYSTEM-TRIGGER system relation. When the action-clause of a trigger is evaluated, the global variable or variables contain copies of exactly the update tuples which satisfy the where-clause of that particular trigger.

Simulating System R Triggers

With this general-purpose trigger mechanism, simulating the behavior of System R triggers in an Extended RTMS database is straightforward. One could duplicate the behavior of the System R triggers in the COMPANY database described earlier, for example, by setting an insert trigger on the EMPLOYEE relation. The where-clause of this trigger would be T. The action-clause of this trigger would be a lisp form which iterates through each tuple in the list **triggered-inserted-tuples**. The insert trigger would have the form:

```
("EMPLOYEE-TRIGGER" "EMPLOYEE" "INSERT" T
 (DO ((INSERTS *TRIGGERED-INSERTED-TUPLES* (CDR INSERTS)))
     ((NULL INSERTS))
   (SETQ INSERTED-TUPLE (CAR INSERTS))
   (MODIFY 'DEPARTMENT 'ATTRIBUTE '(NUMBER-OF-EMPLOYEES)
           'VALUE '((1+ NUMBER-OF-EMPLOYEES))
           'WHERE '(EQUAL DEPARTMENT-NAME
                          ,(THIRD INSERTED-TUPLE))))
 1 T NIL "INSERT TRIGGER TO UPDATE DEPARTMENT RELATION")
```

After an insert into the EMPLOYEE relation, because the trigger where-clause is T, **triggered-inserted-tuples** consists of copies of all of the tuples inserted into the

EMPLOYEE relation. For each inserted tuple, the lisp form action-clause increments the value of the NUMBER-OF-EMPLOYEES attribute of the tuple in the DEPARTMENT relation corresponding to the department of that inserted employee. Delete and modify triggers would be defined analogously.

Simulating Pathfinder Triggers

It is also straightforward, in an Extended RTMS database, to simulate the behavior of triggers in the Pathfinder system. The easiest way to accomplish this is with a properly-defined action-clause. In a military application, for example, one could specify, in the action-clause of insert, delete, and modify triggers on the PLANE relation, that the lists of inserted and/or deleted tuples be sent directly to the display process or stored in certain reserved variables for instant reading by the display process. Thus, a retrieve operation and the corresponding database computation would not be necessary before each new display. An insert trigger to accomplish this task is:

```
("PLANE-TRIGGER" "PLANE" "INSERT" T
 (SETQ DISPLAY-PLANES
       (APPEND *TRIGGERED-INSERTED-TUPLES* DISPLAY-PLANES))
 1 T NIL "INSERT TRIGGER TO UPDATE PLANE DISPLAY")
```

Again, delete and modify triggers are defined analogously.

Alerters

If a user only wishes to be alerted when important updates have occurred, the general-purpose trigger mechanism in Extended RTMS is again quite capable. The user defines the trigger where-clause to specify the condition which must be satisfied by any updates in which he is interested. Then, in the action-clause, he can specify that a flag be set, or a message be sent or printed, so that he will be notified of these important updates. In the military application described above, the user could define the following trigger to inform him when a new unfriendly plane is inserted into the database:

```
("UNFRIENDLY-PLANE-TRIGGER" "PLANE" "INSERT"
 '(EQUAL PLANE-TYPE "UNFRIENDLY")
 (PRINT "A NEW UNFRIENDLY AIRCRAFT IS PRESENT!")
 1 T NIL "INSERT TRIGGER TO NOTIFY ABOUT UNFRIENDLY PLANE UPDATE")
```

Circular Triggers

When defining triggers in Extended RTMS, the user must be careful not to define circular triggers. A circular trigger is one that is set on a relation which, when it fires, causes an update in a second relation which has a trigger set on it, which fires and causes an update in the first relation, firing its trigger, and leading to infinite looping. No checking for circular triggers is implemented in Extended RTMS. The responsibility to avoid circular trigger definition is left to the user.

3.3.2 Cached Views and Incremental Updates

The second real-time mechanism implemented in Extended RTMS addresses the problem of reducing the amount of computation which the database system must perform in order to respond to a posed query. It is too costly for the system to access the base tables each time a retrieve or other query is posed [8]. For this reason, the idea of views has been developed. A view is a table which is derived from the base tables based on some query. When the query is made, the user can specify that the result be stored in the another table, referred to as a materialized view.

In most other traditional relational databases, the updating of this materialized view cannot be done with enough efficiency for real-time applications [3]. When the base tables upon which the view depends change, the view must be brought up to date in an efficient manner in order for the system to exhibit real-time behavior. Complete re-evaluation of the view by examining each tuple in the base tables is generally too inefficient.

Consider a process control database system in a plant that produces several different types of liquor. The plant contains a large number of vats in which the production process is being carried out. The database maintains a VAT relation which contains a

tuple for each vat in the plant. A user might be interested in maintaining a real-time materialized view on the VAT relation. One real-time view which might be of interest is based on the query:

```
(RETRIEVE 'VAT 'WHERE '(EQUAL PRODUCT-TYPE "BEER")
          'PROJECT '(VAT-ID PRODUCT-HEIGHT))
```

The VAT relation currently contains 389 tuples. However, the user is only interested in a small subset of these vats, the ones in which beer is being produced. If not all 389 tuples are updated between view computations, it is desirable not to access the entire base relation each time the new value of this view is required. By focusing on the updates themselves, accesses to the base relation can be avoided.

In [3], a method of efficiently updating materialized views is proposed. This method has not yet been implemented in any existing database system, however. The idea presented is that a materialized view consisting of selections, projections, and/or joins can be updated in a differential or incremental manner, provided that the following is known: the previous value of the view, the states of the base tables before and after all updates have been made, and the set of updates to each of the base tables which define the view. The paper describes a complex graph algorithm of determining, from the set of all updates to the base tables, which updates are relevant to the view. Based on this information, it decides which updates must be deleted from or inserted into the materialized view.

A view cache and incremental update mechanism which incorporates some of these ideas has been implemented in Extended RTMS for join and select views. A similar approach could be used to implement the remainder of the relational operators. This caching mechanism is created when the Extended RTMS system is initialized. It consists of a cache to store the value of each real-time materialized view in the database, as well as a mechanism to incrementally update the cache after updates to the base tables.

Projections are always done last and are not considered to be a part of the view. This convention is followed so that fewer views are stored in the cache. With multiple

cached copies of the same set of tuples projected in various manners, the overhead required to update the cache would become unmanageable.

Rather than using the complex algorithm described in [3] for distinguishing between relevant and irrelevant updates to the base tables, the trigger mechanism is used for this purpose. In fact, the trigger mechanism is given the responsibility of ensuring that each materialized view in the cache is updated properly after an update to the base relation(s) on which the view depends.

The ideas in [3] have also been extended further. It was determined that a user of a real-time database might wish to have the views in which he is interested updated after each update to a base relation, or he might wish to have them updated only when the new view value is explicitly requested. When views are recomputed after each update to a base relation, the time required for each update is increased. The cost of the view computation is distributed among all the base table updates. When views are recomputed only when they are explicitly requested, the time required for base table updates is not increased by as large an amount. This difference is particularly apparent in the recomputation of join views.

Therefore, in Extended RTMS, the user is given a choice of two strategies which may be used to update the cache: active updates and deferred updates. Deferred updates are the default. When active updates are used, after each update to a base table which renders obsolete a materialized view in the cache, that view is recomputed, and the new value is written over the old value in the cache. When deferred updates are used, copies of the update tuples are written to a second (update) cache. Only when the user explicitly requests the current value of that view is the new value calculated and stored in the view cache. In both cases, the new view value is calculated incrementally from the cached previous view value, the set of base-table updates, and, only in the case of a join, the current base tables.

Active Incremental Retrieve Updates

If the user decides that all real-time materialized views should be updated immediately upon updates to the base tables, cached retrieve updates are active. When a RETRIEVE is performed with the cache keyword specified, the value of the RETRIEVE is computed as it normally would be. Before the query result is returned, a copy is cached based on the relation name and the selection criteria. An insert, a delete, and a modify system trigger are defined on the relation for cache maintenance. The where-clause of each trigger is the same as the selection criteria of the RETRIEVE, and the action-clause of each trigger is a lisp form which, when evaluated, updates the cache based on the list **triggered-deleted-tuples** and/or the list **triggered-inserted-tuples**.

Whenever an update is made to the base relation, the trigger of that update type which maintains the retrieve view is retrieved from the SYSTEM-TRIGGER system relation. The where-clause of the trigger, which is the same as the retrieve selection criteria, is evaluated on the list of updated tuples. The result is a list of tuples which are relevant to the view. If the update is an insert, the relevant tuples are added to the list of tuples in the cache corresponding to that retrieve view. On deletes, relevant tuples are deleted from the retrieve view. On modifies, relevant pre-modification tuples are deleted from the retrieve view and relevant post-modification tuples are inserted into the retrieve view. Thus, the cached views are updated incrementally.

The next time the same RETRIEVE query is made, the value is simply read from the cache. The base table is not accessed at all and no further computation must be done, unless a projection is required.

Deferred Incremental Retrieve Updates

If the user decides that all real-time materialized views should be updated only when the new view value is explicitly requested, cached retrieve updates are deferred. Deferred retrieves are very similar to active retrieves. However, the trigger action clauses are defined so that copies of updates to the base table are merely written to an

update cache. When the same RETRIEVE query is next made, the cached copies of the update tuples are retrieved. Relevant updates are determined and tuples are deleted from and/or inserted into the previously cached retrieve view to incrementally form the updated retrieve view. This new view value is then returned as the result of the posed query, after any required projections are performed.

The previous problem of efficiently computing the current value of the real-time materialized view on the VAT relation is now easily resolved. A cached view is defined when the RETRIEVE is first requested, so that the result of the query is cached. Either active or deferred updates can be used. System-defined triggers are set on the VAT relation with a trigger where-clause of '(EQUAL PRODUCT-TYPE "BEER"). Whenever an update is made to the VAT relation, a trigger tests whether it is relevant to the view, that is, whether it satisfies the trigger where-clause. If so, the action-clause of the trigger updates the view accordingly. The base relation is not accessed again when computing this real-time view value.

Active Incremental Join Updates

If the active cache updating method has been chosen by the user, cached join updates are also active. When the join of two base relations in the database is required, the user may specify that the result of this operation be cached. The value of the join is first computed as it normally would be. Before this result is returned, a copy is cached based on the names of the join relations and the join predicate. An insert, a delete, and a modify system trigger are defined on each relation for cache maintenance. The where-clause of each trigger is T and the action-clause of each trigger is a lisp form which, when evaluated, calls an appropriate function to update the cached join view.

The insert trigger action-clause calls a function which "joins" the list of inserted tuples in one relation to the other base relation, based on the join predicate, and adds these tuples to the previously cached join result. The delete trigger action-clause calls a second function which "joins" the list of deleted tuples in one relation to the other base relation, based on the join predicate, and deletes these tuples from the previously

cached join result.

The modify trigger action-clause calls the second function to "join" the list of deleted (pre-modification) tuples in one relation with the other relation and then delete the resultant tuples from the previously cached join result, then calls the first function which "joins" the list of inserted (post-modification) tuples in the first relation with the other relation and then adds the resultant tuples to the previously cached join result. In the future, a routine which directly modifies the tuples in the view, rather than deleting and then inserting tuples, might prove to be more efficient than this method.

The next time the same JOIN view value is required, it is simply read from the cache. The base tables are not accessed and no further computation must be done unless a projection is required.

Deferred Incremental Join Updates

If the deferred cache update method has been chosen by the user, cached join updates are also deferred. Deferred joins are similar to active joins. However, the trigger action clauses are defined so that base table updates are written to the update cache. When the current JOIN view value is required, the following is done:

```
(1) Tuples inserted into relation 1 are "joined" with relation 2
(2) Tuples inserted into relation 2 are "joined" with relation 1
(3) Tuples inserted into relation 1 are "joined" with
    tuples deleted from relation 2
(4) Tuples deleted from  relation 1 are "joined" with
    tuples inserted into relation 2
(5) Resultant tuples of steps 1 thru 4 are added to
    previous join result to form intermediate join result
(6) Tuples deleted from relation 1 are "joined" with relation 2
(7) Tuples deleted from relation 2 are "joined" with relation 1
(8) Tuples inserted into relation 1 are "joined" with
    tuples inserted into relation 2
(9) Tuples deleted from relation 1 are "joined" with
    tuples deleted from relation 2
(10) Resultant tuples of steps 6 thru 9 are deleted from
     intermediate join result to form updated join result
(11) Updated join result is stored in join cache and,
```

```
after any required projections, is
returned to user as result of required join
```

In each "join" step above, the join where-clause is the same as that of the original join.

3.3.3 Real-Time Fields

In many instances, it is simply not practical to update the database using conventional database update operations. If the value of an attribute in some relation changes rapidly, during the process of performing an RTMS:MODIFY routine, the data value may change again. The standard modify routine is too slow to handle rapid updates. The database is not real-time if it cannot properly perform updates within the time constraints set by the environmental changes which are occurring in the system.

Extended RTMS includes a mechanism to bypass the usual modify routine in order to solve this problem. This mechanism gives a user or process the ability to define named pointers into attribute values of the database which are changing rapidly. With these pointers and two new routines, the user can quickly read or update the data values referenced by named pointers. A field which has a direct pointer into it is called a real-time field. The real-time field mechanism addresses a third idea of achieving real-time behavior in a database system, that of decreasing the time required to perform updates to database base tables.

In the previously mentioned ANALYST database system, a stock market analyst can define real-time fields on (set direct pointers into data values of) the CURRENT-STOCK-PRICE attribute of the companies in which he is interested. In particular, he can define a real-time field on the stock price of the companies in which his clients have invested. When the monitor process receives updates from the NYSE, it simply calls the real-time field update routine to rapidly update the stock price of each company to its new value. After the new values are stored in the database, the display process can read and display the updated values. The pointers allow the display process to share memory with the database [5].

Speed vs. Control

Real-time fields address another important issue in database design strategy, that of speed versus control. When using the conventional database update operation to modify a data value, speed is sacrificed for control. The user is not given a handle on the actual data in the database. His interaction is always through the interface and results are returned in the form of copies of the actual data in the relations. Values cannot be changed unless the database is aware of the changes. The user can be prevented from inserting an illegal data value into the database. Parameter checking and error checking are available. However, all these services are provided at a cost, computation time.

If the user is given a direct pointer to the actual data value stored in an Extended RTMS database, database control is given up. The user is allowed to change this value as he desires. Thus, a significant amount of responsibility is transferred from the system to the user. However, a great improvement in speed is gained. A data value can be modified very quickly. As long as the user is careful to use only legal attribute values, there is no problem.

When real-time fields are defined on a relation, Extended RTMS ensures that no indices may be defined on this relation. This restriction is enforced because the modification of a data value through a direct pointer could invalidate indices defined on the relation. For the same reason, if indices are already defined on a relation, direct pointers cannot be set into that relation. Similarly, real-time fields cannot be defined on a relation which has had a RETRIEVE or JOIN result cached. Modifying a data value using a direct pointer would invalidate the cache.

Update Sampling

If the sensors monitoring a real-time quantity are receiving updates at an extremely rapid rate, a rate even greater than the rate at which updates can be made through the real-time field mechanism, then some sort of compromise must be reached.

Perhaps the best solution to this problem is to simply set an update sampling frequency which will allow for one update to be processed before the next is sampled. Some data will be lost with this strategy but the database will have enough time to complete each individual update.

3.3.4 Real-time Triggers

As a final extension of RTMS, the trigger mechanism and the real-time field mechanism were integrated. It is likely that a user of the real-time field mechanism would want to define triggers on modifications to a relation through the real-time field update routine. Real-time triggers allow for this functionality. A real-time trigger is a modify trigger which has a reserved priority value of 2.

The user specifies, with an environment variable, whether or not real-time triggers should be executed. If so, whenever a modification is made to a real-time field value, the real-time triggers set on the relation are retrieved. The trigger where-clause is evaluated on the pre-modification tuple and on the post-modification tuple. If the result of either of these evaluations is non-null, the action-clause of the trigger is evaluated. The real-time trigger may take a different action for the pre-modification tuple than for the post-modification tuple. If the user has not specified that real-time triggers should be executed, then no attempt is made to retrieve triggers from the SYSTEM-TRIGGER system relation. In this situation, the overhead of trigger retrieval is eliminated.

Chapter 4

Analysis of Real-time Mechanisms in Extended RTMS

4.1 Methodology

The previous chapter described general-purpose triggers, the view cache with incremental updates, real-time fields, and real-time triggers, all of the mechanisms which were implemented in Extended RTMS. Each of these mechanisms was expected to enhance the functionality and/or performance of RTMS so that real-time applications can be built on top of an Extended RTMS database. This chapter analyzes these real-time mechanisms. The time and/or computation required to perform corresponding operations in conventional RTMS and Extended RTMS is compared.

Triggers, real-time triggers and real-time fields significantly affect the performance of an RTMS database and allow for real-time applications to be built with an Extended RTMS database. When analyzing these mechanisms, times required to perform operations in conventional RTMS and Extended RTMS are compared. The recorded times were averaged over several trials to control for interrupt and data clustering effects. In each case, the times recorded for the several trials were very close. This indicates that undesired effects were eliminated or made uniform. Incremental garbage collection was turned off to eliminate its effects on these times.

As currently implemented, RTMS:RETRIEVE and RTMS:JOIN contain a sub-

stantial amount of overhead. This overhead is seen in both conventional RTMS and Extended RTMS. Its effects overshadow the savings in time which incremental view computation yields, making a comparison of the absolute times required to compute RETRIEVE views and JOIN views in conventional RTMS and in Extended RTMS less meaningful. A real-time measure which is more meaningful, and is used in this case, is a comparison of the number of basic operations that must be performed to compute the view in each system. Basic operations consist of tuple evaluations, tuple comparisons, and low-level retrievals that are performed when computing the view. Also, the percentages of increase in time required to perform base table updates are given to show the relative overhead of the various mechanisms.

In addition to the overhead inherited from conventional RTMS, Extended RTMS has two new sources of overhead that need to be explained. These are discussed first.

4.2 Extended RTMS Overhead

The two most significant new sources of overhead in Extended RTMS are found in the trigger mechanism. Each time an update is performed on a relation in an Extended RTMS database, the active triggers, of that update type, defined on the relation, are retrieved with a QTRIEVE call. QTRIEVE, short for quick retrieve, is used to retrieve tuples from all system relations. A QTRIEVE call requires a minimum of .22 seconds compared with a minimum of .87 seconds required for a RETRIEVE. These numbers correspond to a retrieval from a one tuple relation based on a non-trivial where-clause.

After a trigger is retrieved, the trigger where-clause is evaluated on the list of update tuples. One EVAL-WHERE function call performs this where-clause evaluation. The overhead which accompanies an EVAL-WHERE call is .24 seconds. Therefore, each update is subjected to a total overhead of .46 seconds due to the trigger mechanism. This overhead should be eliminated before Extended RTMS is used in a commercial real-time system.

Nearly all the overhead of the QTRIEVE operation can be eliminated if the re-

lations on which triggers are defined are implemented as flavor instances. Then triggers are defined as the after-methods of flavors. Only .0015 seconds of overhead accompanies a method invocation as compared with the .22 seconds needed to retrieve triggers from the SYSTEM-TRIGGER relation.

In order for triggers to be defined as after-methods, however, all relations on which the triggers are defined would be required to store their tuples as flavor instances. This restriction would remove a great deal of the flexibility afforded by the RTMS system. For this reason, the former approach was chosen. After Common Objects are implemented in CommonLisp, RTMS relations will be implemented as Common Objects. Then after-methods can be used to implement triggers without a loss in RTMS flexibility, and the overhead of a QTRIEVE will be eliminated.

If the evaluation of a trigger where-clause predicate on the list of update tuples yields a non-null result, then the action-clause of the trigger must be evaluated. The average time required for the evaluation of a trigger action-clause is not quantifiable since the trigger action-clause can be any arbitrary lisp form. If the action-clause specifies that a variable value be reset, its evaluation time is only .001 seconds. If the action-clause specifies that a message be printed on a display, the evaluation time is .01 seconds or more.

If, however, the action-clause specifies that a large body of code be executed, the evaluation time could be very long. In this case, the real-time behavior of the system could be lost. The user is given the responsibility of ensuring that the action-clause of each trigger is defined appropriately, so that real-time behavior, as defined by the particular application, is maintained.

4.3 Analysis of General-Purpose Triggers

4.3.1 Simulating System R Triggers

When Extended RTMS triggers are used to simulate the behavior of System R triggers, it is not clear that a performance gain is achieved. The gain in this case is

more of a gain in functionality. Constraints between relations can be enforced through the trigger mechanism. The user will not have to explicitly make a second modify call after modifying a value in one relation on which another relation depends. He simply sets a trigger on the first relation with an action-clause that calls the MODIFY routine to update the second relation.

4.3.2 Simulating Pathfinder Triggers

However, when Extended RTMS triggers are used to simulate the behavior intended for Pathfinder triggers, a noticeable performance gain is seen. Consider the process control database again. Updates to the VAT relation are of the following form:

```
(INSERT 'VAT 'TUPLES list-of-tuples)

(DELETE-TUPLES 'VAT 'WHERE selection-criteria)

(MODIFY 'VAT 'WHERE selection-criteria
        'ATTRIBUTE attribute-list
        'VALUE value-list)
```

In a conventional RTMS database system, a display process has no information about the updates that are performed to the process control database. In order to maintain a display of the height of beer in the beer vats, therefore, the following query must be made before each redisplay:

```
(RETRIEVE 'VAT 'WHERE '(EQUAL PRODUCT-TYPE "BEER")
          'PROJECT '(VAT-ID PRODUCT-HEIGHT))
```

Thus, the display process must poll the 389 tuple VAT relation, performing periodic retrievals to get the current value of this view. Each retrieval requires an average of 1.28 seconds. If, since the last display, no updates have been made to the vats which contain beer, then the retrieval yields no new information.

In Extended RTMS, triggers are used to eliminate these retrievals. The following insert trigger is set on the VAT relation:

```
("BEER-VAT-TRIGGER" "VAT" "INSERT" '(EQUAL PRODUCT-TYPE "BEER")
 (SEND BEER-VATS-DISPLAY :ADD-TO-DISPLAY
                         *TRIGGERED-INSERTED-TUPLES*)
 1 T NIL "INSERT TRIGGER TO UPDATE BEER VAT DISPLAY")
```

On a relevant update, the appropriate trigger sends a message to the object BEER-VATS-DISPLAY, which updates the display with the inserted tuples. Delete and modify triggers are defined similarly.

With these triggers defined on the VAT relation, the display process knows that no relevant updates have been made unless it receives a redisplay message. If no message is sent, then no redisplay is necessary. It simply maintains its current display. Thus, useless polling of the database is eliminated. Since each RETRIEVE currently requires 1.28 seconds, a substantial savings is seen when using general-purpose triggers in this manner in Extended RTMS.

4.4 Analysis of Incremental Updates

As stated in the approach section, one of the goals of the Extended RTMS system is to minimize the amount of computation required to recompute the value of a frequently accessed database view. It was stated, in [3], that this efficient incremental updating of materialized views would enable a database system to be used for real-time applications. The goal of this section is to determine the amount of computation that is saved when incrementally recomputing a cached database view in Extended RTMS.

This section compares the computation required to incrementally update cached view values after database updates and the computation required to recompute the views from both indexed and non-indexed base relations. The performance of active and deferred updates to the cache is compared. It also contrasts the relative overhead which the various mechanisms add to base table updates to get an indication of their relative efficiency.

4.4.1 Analysis of Incremental Retrieves

As stated above, the overhead inherent in RTMS:RETRIEVE is too large to allow for a meaningful comparison of the time required to compute retrieve views in conventional RTMS and Extended RTMS. However, by comparing the number of basic operations required to recompute a retrieve view after base table updates in each system, we can get a useful measure of which system "does less work" to compute this views.

Conventional RTMS performs a retrieval from a base relation by calling EVAL-WHERE, which evaluates the where-clause selection criteria on the list of tuples in the base relation. If the base relation is not indexed, all tuples in the relation must be inspected. If it is indexed, this number is reduced. The tuples which satisfy the selection criteria are returned as the retrieve result. The number of tuples which must be passed to EVAL-WHERE, in this case, defines the amount of "work" that conventional RTMS must perform in order to compute the view.

Suppose RELA is a non-indexed relation of m tuples from which a retrieve has been made. If x tuples are inserted into RELA, the new cardinality of RELA is $(m+x)$. To retrieve from RELA in conventional RTMS, $(m+x)$ tuples must be passed to EVAL-WHERE so the selection criteria can be tested on them. If y tuples are deleted from RELA, the new cardinality of RELA is $(m-y)$. On a retrieve call, these $(m-y)$ tuples must be passed to EVAL-WHERE and tested. After z modifications are made in RELA, then the cardinality of RELA remains at m. Then m tuples must be passed to EVAL-WHERE.

In Extended RTMS, these numbers can be reduced. If a retrieve is performed on RELA, which contains m tuples, all m tuples must be passed to EVAL-WHERE the first time only. Then the view is cached. When active updates are used, after an INSERT of x tuples in RELA, only these x tuples are passed to EVAL-WHERE which determines if they are relevant to the view. Since x is always less than $(m+x)$, the number of basic operations, tuples inspected by EVAL-WHERE in this case, is always decreased. On an DELETE of y tuples from RELA, only these y tuples are passed to EVAL-WHERE. When $y < (m-y)$, the number of basic computations is decreased.

On a MODIFY of z tuples, 2z tuples are sent to EVAL-WHERE which determines if they are relevant. Thus, if $2z < m$, the computation is decreased.

When cache updating is deferred, several different updates may be performed before the retrieve view is explicitly required. If x tuples are inserted, y tuples are deleted, and z tuples are modified in RELA, conventional RTMS inspects $(m + x - y)$ tuples when recomputing the view. Extended RTMS, inspects $(x + y + 2z)$ tuples in order to update the cached view. The number of tuples in the base relation RELA, m, generally dominates over the number of update tuples. Thus, in most cases, $(x + y + 2z) < (m + x - y)$ and deferred incremental cache updates require less computation than complete recomputation of the view from the base table.

A comparison has been made of the number of tuples examined when computing a retrieve view after updates to the base table in conventional RTMS and Extended RTMS. Cached retrieves with deferred and active updates are compared with retrieves from a non-indexed base relation and an indexed base relation. The percentage of overhead added to the updates by the incremental view calculation in Extended RTMS is compared with that added to the updates by the hashed index in conventional RTMS. The VAT relation, initially containing 389 tuples, was used, and the view examined was the following:

```
(RETRIEVE 'VAT 'WHERE '(EQUAL PRODUCT-TYPE "BEER"))
```

Updates to the VAT relation consisted of modifies of 14, 39, and 142 tuples, deletions of 14, 39, and 128 tuples, and insertions of 5 tuples. The results are shown in Tables 4.1, 4.2, 4.3, and 4.4.

The results are interesting in several ways. First, computation of the view from the non-indexed base relation requires that all tuples in the base relation are inspected. Cached views with both deferred and active updates require significantly less computation (fewer tuples sent to EVAL-WHERE) than the non-indexed relation to compute the new view value. Notice that for all cached views, only the update tuples are passed to EVAL-WHERE once the view has been initially cached.

Non-indexed Retrieve			
Update Type	# Tuples Updated	# Tuples Inspected	% Overhead
Modify	14	389	–
Modify	142	389	–
Modify	39	389	–
Delete	14	375	–
Delete	128	247	–
Delete	39	208	–
Insert	5	213	–

Table 4.1: Retrieve View Recomputation from Non-indexed Relation

Hashed Index Retrieve			
Update Type	# Tuples Updated	# Tuples Inspected	% Overhead
Modify	14	389	141.5
Modify	142	389	81.7
Modify	39	389	128.0
Delete	14	14	134.9
Delete	128	128	141.3
Delete	39	39	117.8
Insert	5	5	40.9

Table 4.2: Retrieve View Recomputation from Indexed Relation

Cached Retrieve with Deferred Updates			
Update Type	# Tuples Updated	# Tuples Inspected	% Overhead
Modify	14	28	24.4
Modify	142	284	17.2
Modify	39	78	24.4
Delete	14	14	49.7
Delete	128	128	46.7
Delete	39	39	47.4
Insert	5	5	76.3

Table 4.3: Cached Retrieve View Recomputation with Deferred Updates

Cached Retrieve with Active Updates			
Update Type	# Tuples Updated	# Tuples Inspected	% Overhead
Modify	14	28	61.2
Modify	142	284	45.8
Modify	39	78	57.1
Delete	14	14	59.1
Delete	128	128	74.0
Delete	39	39	69.6
Insert	5	5	95.7

Table 4.4: Cached Retrieve View Recomputation with Active Updates

After an RTMS hash table is used to index the relation by PRODUCT-TYPE, only the update tuples on an insert or a delete are examined. This is also the case in Extended RTMS. After a modify, however, conventional RTMS clears the hash table and rehashes the base relation. All tuples in the base relation are examined for this step. Extended RTMS only inspects the pre-modification update tuples and the post-modification update tuples to update the cache.

In many ways, a hashed index and the view cache are similar for retrieve views. The index hashes a relation on an attribute value while the cache hashes a view by where-clause value. In both cases, the proper tuples can be retrieved immediately using the hash table key. However, the percentage of overhead added to base table updates by the index is seen to be much larger than the overhead added by incremental updates in all cases except the insertion of 5 tuples. When a larger number of tuples are updated, cached views with incremental updates are preferable over hashed secondary indices.

There is a significant difference in the amount of overhead added by deferred and active updates. When deferred updates are used, copies of update tuples are simply written to an update cache. Relevant tuples are not determined until the view is explicitly requested. Thus, the time required to compute these relevant updates is not seen as a part of the time required to perform the base table updates. It is seen as a part of the time required to recompute the retrieve view.

4.4.2 Analysis of Incremental Joins

As with RTMS:RETRIEVE, the overhead inherent in RTMS:JOIN is too large to allow for a meaningful comparison of the time required to compute join views in conventional RTMS and Extended RTMS. Therefore, the number of basic operations required to recompute join views after base table updates is compared to get a notion of which system "does less work" to compute these views. For join views, measures of basic computations are the number of tuples inspected and the number of where-clause pre-processing steps and low-level retrievals made.

A join of two base relations in RTMS is one of two types, a theta-join or an equi-join. Although an equi-join is actually a special case of a theta-join, in this paper, the term theta-join will refer to a non-equi-join. Theta-joins are examined first.

Theta-joins

When conventional RTMS performs a theta-join of two relations, it retrieves the smaller relation first. Then it iterates through each of the tuples in this relation. For each tuple in the smaller relation, in order to execute the join predicate, the join where-clause is pre-processed and then a low-level retrieval is made into the larger relation based on the intermediate where-clause. The tuple from the smaller relation is appended to each tuple in the larger relation which satisfies the intermediate where-clause, forming the "join" of that tuple in the smaller relation with the larger relation. Once all tuples in the smaller relation have been iterated through, the resultant tuples of each of the "joins" are combined into a list and returned as the join result.

This procedure requires one fixed retrieve from the smaller relation, plus one low-level retrieval from the larger relation for each tuple in the smaller relation. Consider two base relations RELA and RELB of lengths m and n respectively. After an insertion of x tuples into RELA, RELA contains $(m + x)$ tuples. In order to join these two relations, $min((m + x), n)$ where-clause pre-processing steps must be performed and $min((m + x), n)$ low-level retrievals from the larger relation based on the pre-processed

where-clause must be performed. Each low-level retrieval from the larger relation must inspect, using EVAL-WHERE, $max((m + x), n)$ tuples. $O((m + x) * n)$ tuples in total must be inspected in order to compute the view.

After a deletion of y tuples from RELA, $min((m - y), n)$ where-clause pre-processing steps and low-level retrievals from the larger relation are performed. Each low-level retrieval from the larger relation inspects $max((m-y), n)$ tuples. $O((m-y)*n)$ tuples are inspected in total.

A modification in RELA will not affect the number of tuples in the relation. Therefore, $min(m, n)$ where-clause pre-processing steps and low-level retrievals from from the larger relation are performed. Each low-level retrieval from the larger relation inspects $max(m, n)$ tuples, and $O(mn)$ tuples are inspected in total.

When cached joins and active incremental updates are employed in Extended RTMS, these numbers can be reduced. Again, RELA contains m tuples before any updates and RELB contains n tuples before any updates. On an insert or delete of delta tuples in RELA, one fixed retrieve from RELB is made. The lengths of the list of updated tuples and RELB, delta and n respectively, are compared. Then, $min(delta, n)$ where-clause pre-processing steps are performed and $min(delta, n)$ low-level retrievals from the larger of the two lists based on the pre-processed where-clause are performed. Low-level retrievals from the larger list inspect, using EVAL-WHERE, $max(delta, n)$ tuples. Thus, $O(delta * n)$ tuples in total are inspected in order to compute relevant join tuples which are appended into or deleted from the previously cached view value.

After an insertion of x tuples into RELA, fewer tuples, in total, are always inspected when updating the view using incremental updates since x is always less than $(m + x)$. If $x < n$ also, then fewer where-clause pre-processing steps and fewer low-level retrievals are performed as well. On a deletion of y tuples from RELA, if $y < (m - y)$, fewer tuples, in total, are inspected using incremental updates. If $y < (min(m - y), n)$ then fewer where-clause pre-processing steps and fewer low-level retrievals are performed as well.

On a modification of z tuples in RELA, RELB is retrieved. An incremental

Conventional Theta-join				
Update Type	# Tuples Updated	# Low Level Retrievals	Total # Tuples Inspected	% Overhead
Modify	14	39	15,171	-
Modify	142	39	15,171	-
Modify	39	39	15,171	-
Delete	14	39	14,625	-
Delete	128	39	9,633	-
Delete	39	39	8,112	-
Insert	5	39	8,307	-

Table 4.5: Conventional Theta-join View Recomputation

update after a modification is performed by calling an incremental delete update routine followed by an incremental insert update routine. Thus, $2 * min(z, n)$ where-clause pre-processing steps and $2 * min(z, n)$ low-level retrievals based on the pre-processed where-clause are performed. $2 * max(z, n)$ tuples in the larger list are inspected using EVAL-WHERE and $O(2zn)$ tuples, in total, are inspected in order to update the cached join view.

Provided that $2z < m$, fewer tuples are inspected after a modify when updating the view using incremental updates. If $2z < min(m, n)$ also, then fewer where-clause pre-processing steps and fewer low-level retrievals are performed as well. In each of these cases, the argument is symmetric for an update in RELB.

The following theta-join of two relations in the process control database was computed in conventional RTMS and in Extended RTMS:

```
(JOIN 'FROM '(VAT ROBOT) 'WHERE
      '(AND (PRODUCES ROBOT.TYPE VAT.PRODUCT-TYPE)
            (EQUAL VAT.PRODUCT-TYPE "BEER"))
```

The VAT relation initially contained 389 tuples and the ROBOT relation contained 39 tuples. Tables 4.5, 4.6, and 4.7 are a comparison of the number of tuples inspected and low-level retrievals performed when computing this theta-join view value after various updates to the base relations.

It is seen that the total "work" done (the total number of tuples inspected) when

Cached Theta-join with Deferred Updates				
Update Type	# Tuples Updated	# Low Level Retrievals	Total # Tuples Inspected	% Overhead
Modify	14	28	1,092	27.9
Modify	142	78	11,076	16.8
Modify	39	78	3,042	25.0
Delete	14	14	546	43.0
Delete	128	39	4,992	41.3
Delete	39	39	1,521	46.7
Insert	5	5	195	74.2

Table 4.6: Cached Theta-join View Recomputation with Deferred Updates

Cached Theta-join with Active Updates				
Update Type	# Tuples Updated	# Low Level Retrievals	Total # Tuples Inspected	% Overhead
Modify	14	28	1,092	1,099
Modify	142	78	11,076	1,718
Modify	39	78	3,042	2,048
Delete	14	14	546	572
Delete	128	39	4,992	1,475
Delete	39	39	1,521	1,302
Insert	5	5	195	387

Table 4.7: Cached Theta-join View Recomputation with Active Updates

computing the cached view while using either deferred or active incremental updates is far less than the total "work" done when recomputing this view from the base tables. After insertions and deletions, the number of low-level retrievals performed is either maintained or decreased in Extended RTMS. However, after the modifications of 39 and 142 tuples, twice as many low-level retrievals are performed. This is true because one "join" must be performed to determine which tuples to delete from the previously cached view, and a second "join" must be performed to determine which tuples to append to this cached view. Again, a routine which directly modified the tuples in the view, rather than deleting and then inserting tuples, might save computation here.

The number of tuples examined and the number of low-level retrievals performed to update the cache is the same for deferred and active updates. However, the overhead added to base tables updates by active updates is tremendous. Deferred updates are usually preferable to active updates, since they are much more transparent to a user of the database system. With deferred cache updates, the join view computation is done only when explicitly requested. Therefore, much less overhead is added to the base table updates.

Equi-joins

For an equi-join of RELA and RELB, the situation is quite different. Conventional RTMS retrieves both relations and then hashes the larger relation. All tuples in the larger relation are inspected for this step. Then it iterates through each tuple in the smaller relation, retrieving from the hash, the bucket of tuples which satisfy the equi-join where-clause predicate. The individual tuples in the bucket are guaranteed by hashing to satisfy the equi-join where clause. Therefore, they are not inspected again at this step. The tuple from the smaller relation is simply appended to each tuple in the bucket. The resultant tuples from each hash table retrieve-and-append are combined in a list of tuples and returned as the result of the join operation.

In this procedure, $max(m, n)$ tuples are inspected to hash the larger relation and $min(m, n)$ hash table retrievals are made to determine the buckets of tuples satisfying

the equi-join where-clause. Thus, $O(m+n)$ operations, in total, are performed in order to compute the view from the base tables. After an insertion, deletion, or modification into RELA, $O(m+delta+n)$, $O(m-delta+n)$, and $O(m+n)$ operations, respectively, are performed to recompute the view.

In Extended RTMS, caching with deferred updates is not used on equi-joins. This convention is followed because on updates to both relations, each relation would have to be hashed and then retrieved from. So caching with deferred updates is more computation intensive than simply recomputing the view from the base tables. Thus, only active updates are used with cached equi-joins.

When an equi-join of RELA and RELB is cached, active updates to the cache after an insert or delete of delta tuples in RELA are performed in the following manner. RELB is retrieved and the larger of RELB and the list of update tuples in RELA is hashed. $Max(delta, n)$ tuples are inspected for this step. Then, for each tuple in the smaller list, the bucket which satisfies the equi-join where-clause predicate is retrieved from the hash table, and the tuple is appended to each tuple in this bucket. $Min(delta, n)$ hash table retrieves are required for this step. Then the resultant tuples are appended into or deleted from the previously cached view value. Thus, $O(delta+n)$ operations, in total, are performed in order to compute the updated view value using active incremental updates. For a modification update, which calls a delete update routine then an insert update routine, these counts are doubled.

Since x is always less than $(m+x)$, the total number of operations on an insert of x tuples in Extended RTMS is always less than that of conventional RTMS. If $y < (m-y)$ on a deletion of y tuples, and if $2 * (z+n) < (m+n)$, on a modification of z tuples, then fewer operations are performed in Extended RTMS cached equi-join view updates after deletes and modifies also.

The following equi-join was computed in conventional RTMS and in Extended RTMS:

```
(JOIN 'FROM '(VAT ROBOT) 'WHERE '(EQUAL PRODUCT-TYPE PRODUCT))
```

Conventional Equi-join			
Update Type	# Tuples Updated	Total # Operations	% Overhead
Modify	14	428	–
Modify	142	428	–
Modify	39	428	–
Delete	14	414	–
Delete	128	286	–
Delete	39	247	–
Insert	5	252	–

Table 4.8: Conventional Equi-join View Recomputation

Cached Equi-join with Active Updates			
Update Type	# Tuples Updated	Total # Operations	% Overhead
Modify	14	106	119.7
Modify	142	362	313.0
Modify	39	156	174.4
Delete	14	53	131.5
Delete	128	167	455.3
Delete	39	78	158.5
Insert	5	44	119.4

Table 4.9: Cached Equi-join View Recomputation with Active Updates

Again, the VAT relation initially contained 389 tuples and the TEAM relation contained 39 tuples. Tables 4.8 and 4.9 are a comparison of the total number of operations performed when computing this equi-join view value after updates to the base relations.

Note that the total "work" done to compute the theta-join view values described in the previous section is more than an order of magnitude greater than the total "work" done to compute these equi-join view values.

In all seven cases, the total number of operations performed when recomputing the equi-join view is smaller in Extended RTMS than in conventional RTMS. The savings on inserts and deletes is more pronounced than the savings on modifies. The percentage of overhead added to base table updates by the active update mechanism is again large. However, this percentage is much smaller than that added to base table updates when active updates are used to recompute theta-join view values, which

require significantly more time and computation.

4.5 Analysis of Real-time Fields

As stated previously, real-time fields give an Extended RTMS user the ability to define pointers directly into the database fields where the values of attributes are actually stored. A user can examine the value of a real-time field directly with the routine GET-REAL-TIME-FIELD, and can set this value directly with the routine RESET-REAL-TIME-FIELD. This section examines the amount of time saved when reading and updating a data value with the real-time field mechanism.

The RTMS:MODIFY routine is the conventional method for updating an attribute value in RTMS. In order to update the value of one attribute of one tuple in the unindexed 389 tuple VAT relation using the following routine,

```
(MODIFY 'VAT 'ATTRIBUTE '(PRODUCT-HEIGHT) 'VALUE '(35.02)
        'WHERE '(EQUAL VAT-ID 77))
```

an average of 1.29 seconds is required. In a real-time environment in which sensors are monitoring dynamic objects and are updating attributes with explicit values, each update would have to be performed in this manner. One tuple would be updated per RTMS:MODIFY call. Thus, only one tuple can be updated every 1.29 seconds.

The following query is used to retrieve the vats in which beer is being made and the height of beer in each vat:

```
(RETRIEVE 'VAT 'WHERE '(EQUAL PRODUCT-TYPE "BEER")
          'PROJECT '(VAT-ID PRODUCT-HEIGHT))
```

This RETRIEVE requires an average of 1.28 seconds in Conventional RTMS. Therefore, the total time required to update one tuple in the view and then retrieve the new view value is 2.57 seconds. Since many tuples in a relation with a real-time attribute will need to be modified rapidly, it is clear that conventional RTMS cannot be used in such a real-time environment.

However, if real-time fields are used, real-time behavior is possible. A real-time field is defined on the PRODUCT-HEIGHT attribute of the tuple corresponding to VAT#77. In order to update this attribute value using RESET-REAL-TIME-FIELD, only .01 seconds are required if no real-time-triggers are set on the VAT relation. So 100 tuples can be directly updated per second using real-time fields. This is a substantial improvement over the performance achievable using MODIFY.

GET-REAL-TIME-FIELD can be used to retrieve the attribute value into which a direct pointer is set in an average of .0006 seconds. So the values of 1667 real-time fields can be retrieved in a second using this routine. The number of real-time fields that is defined does not significantly affect the time required for any individual RESET-REAL-TIME-FIELD operation or GET-REAL-TIME-FIELD operation, since a pointer is merely being reset or evaluated in each case. If the user wants the value of the entire view, it can be retrieved in 1.28 seconds using RETRIEVE.

Thus, in a real-time environment in which the value of a real-time attribute of each tuple in a relation must be updated individually, the performance of conventional RTMS is not adequate. Extended RTMS with real-time fields, however, is well suited for this task.

4.6 Analysis of Real-time Triggers

Real-time triggers set on a relation in which real-time fields are defined allow for important updates to be recognized and acted upon. However, they also have an impact on the time required to perform all updates to that relation.

As with ordinary modify triggers, overhead due to the real-time trigger mechanism is the following: the time required to QTRIEVE the real-time triggers from the SYSTEM-TRIGGER system relation, the time required to perform an EVAL-WHERE of the trigger where-clause on the pre-update tuple and perform an EVAL-WHERE of the trigger where-clause on the post-update tuple, and the time required to evaluate the trigger action-clause if the where-clause evaluation yields a non-null result. When

using real-time triggers, the user is urged to carefully define the action-clause so that real-time behavior is maintained.

If he desires, the user can set a switch to prevent real-time triggers from being retrieved and evaluated. In this case, no trigger overhead will be seen.

When no real-time triggers are defined on a relation in which real-time fields are defined, the field can be updated in .01 seconds. If a real-time trigger is set whose action-clause specifies that a message be written to the screen when an important update has been made, an update to the field requires an average of 1.00 seconds. Thus, real-time triggers can add noticeable overhead to the real-time field mechanism. However, real-time triggers can be used to inform a user or display process about important updates to the database through the real-time field mechanism. Used in this way, they eliminate unnecessary retrievals from (polling of) the database in the same manner that ordinary triggers do.

4.7 Military Demo

This section describes a demo which highlights many of the features implemented in the Extended RTMS system. As stated before, many military applications need a general-purpose, real-time database system. One example is an application which maintains, in a database, information on planes which are moving in real-time, and displays these planes in a graphic display.

To demonstrate this capability, a relational database containing relations for runways, airports, areas, and planes was built using Extended RTMS. Bitmapped background maps were integrated into the system for display purposes. The goal of the demo was to be able to update the position of a plane in the database and redisplay it in its correct new position on the map in real-time. Only one plane, the plane that would be flown remotely, was stored in the PLANE relation and updated in this scenario. The extension to multiple moving planes is fairly straightforward, although, due to an inefficient display routine, harder to make real-time.

First, the relational database was implemented in conventional RTMS to see what performance would be achievable without the new capabilities of Extended RTMS. A conventional RTMS:MODIFY was used to update the position of the plane. The update process took an average of .82 seconds. Then the new plane position value had to be retrieved from the PLANE relation. An RTMS:RETRIEVE operation, requiring .87 seconds, performed this retrieval. After the plane's new position was retrieved, it was displayed, requiring an average of .93 seconds. Thus, the entire procedure of modifying the plane's position in the database, retrieving the new value, and redisplaying the plane on the map required 2.62 seconds. This amount of time is not acceptable in a dynamic, real-time environment such as this.

Next, the database was built onto Extended RTMS. The relations did not require any modification in order to create the Extended RTMS database. A real-time field was defined on the POSITION attribute of the remotely-flown plane in the PLANE relation and a real-time trigger was set on the PLANE relation. The action-clause of this real-time trigger specified that the new value of the plane's position be written to a global variable *PLANE-POSITION*.

The new demo scenario is as follows. The position of the remotely flown plane is updated directly through the real-time field defined on the plane. This update to the plane's position triggers the update of the global variable *PLANE-POSITION* which also holds the value of the plane's position. Once this update has been made, the plane's new position must be displayed. To redisplay the plane, the new position is simply read from the global variable *PLANE-POSITION* and then the plane is displayed in this new position.

This new procedure, using Extended RTMS, is noticeably faster. The database update of the plane's position, including the real-time trigger retrieval and evaluation and the update of the global variable *PLANE-POSITION*, now requires an average of 1.07 seconds. .22 seconds of this time is overhead spent in retrieval of the trigger. However, the reading of the new plane position value from the variable *PLANE-POSITION* takes an average of 110 microseconds, the variable is merely evaluated. The redisplay of the plane on the background map is unchanged. An icon is written

onto the background map at the specified plane position. Again this requires an average of .93 seconds.

The total time for the procedure is reduced to 2.00 seconds, from the 2.62 seconds in conventional RTMS. Two seconds is closer to a real-time response for this update and redisplay procedure and is acceptable for the particular application described. The time is saved in the retrieval of the plane's new position by writing it to an easily accessible variable through the use of a real-time trigger. If real-time triggers are implemented as methods in the future, the trigger overhead will be eliminated and the time savings will be even greater.

This demo showed that Extended RTMS can be instrumental in enabling the real-time redisplaying of a dynamic object, in this case a plane. The real-time trigger mechanism in Extended RTMS was also used for another purpose in this demo. One could imagine that air traffic controllers or others monitoring a display of aircraft would be interested in knowing when an aircraft has crossed into, or out of, their airspace.

A second real-time trigger was defined on the position of the remotely-flown plane. The action-clause of this trigger tests, after the plane's position has been updated, whether the plane is within the area of control of a particular airport. If the plane's last position was outside the airport's area of control but its new position is within the airport's area of control, the message,

```
"REMOTE-PLANE HAS ENTERED airport-name AREA"
```

is printed to the screen and the plane icon is made white. This signifies that the plane is now of interest to an air traffic controller monitoring that area. If the plane's last position was within the airport's area of control but its new position is outside the airport's area of control, the message,

```
"REMOTE-PLANE HAS EXITED airport-name AREA"
```

is printed to the screen and the plane icon is made black. This signifies that the plane is no longer of interest to an air traffic controller monitoring that area. When employed

in this manner, real-time triggers are seen to be useful for alerting a user when an important update has been made to a real-time field in the database.

Chapter 5

Conclusions and Future Research

5.1 Conclusions

This thesis shows that software extensions can be made to an existing general-purpose database management system to enable applications which process real-time data to maintain it in a database. The real-time application specifically considered here is the real-time monitoring of dynamic objects. The software extensions which have been implemented in an existing relational database system as part of this research are general-purpose triggers, real-time triggers, real-time fields, and a view cache with incremental updates.

General-purpose triggers and real-time triggers give a user or process information about the updates to base tables in the database, and thereby reduce the number of queries that must be made to the database. General-purpose triggers are used with the more conventional insert, delete, and modify routines, while real-time triggers are used with real-time fields. Real-time fields allow a user or process to share memory with the database by maintaining direct pointers into the database tables. With these pointers, data values can be read or updated directly and very rapidly by a user or process.

The view cache with incremental updates reduces the amount of database computation that must be performed to correctly respond to a query. It introduces a new type of query optimization, one based on database updates. The view cache also gives the user the ability to define a view as being a high priority view, one that will be

required often. The user is given a choice of whether this view should be updated immediately upon updates to the base table(s) on which it depends, or updated only when specifically requested.

Real-time fields account for the most significant time savings in Extended RTMS. In a 389 tuple relation in an Extended RTMS database, the time required to reset the value of a real-time field is .01 seconds, and the time required to read this value is .0006 seconds. The corresponding modify and retrieve operations in a 389 tuple relation in conventional RTMS require 1.29 and 1.28 seconds respectively.

Triggers eliminate unnecessary polling of the database by informing a user or process when important updates have occurred. The savings is large since a retrieve from a 1 tuple relation requires .87 seconds and a retrieve from a 389 tuple relation requires 1.28 seconds. However, in Extended RTMS, both types of triggers are subject to the .22 seconds of overhead required to retrieve them from the system relation in which they are stored. When Common Objects are implemented in CommonLisp, relations will be implemented as Common Objects and triggers as after-methods. Then this overhead will be eliminated.

5.2 Future Research Directions

This research has addressed many ideas for extending a database management system to enable real-time applications to maintain rapidly changing data in a database. However, these ideas can be extended even further. This section describes four directions in which the work done here can be extended.

5.2.1 Background Updates

As implemented, the Extended RTMS view cache can be updated using active updates or deferred updates. A third cache update mechanism that could be implemented is a background update mechanism. View updates could be deferred until the database system is not busy performing a base table update or responding to a query.

When no other database computation is being performed, the system incrementally updates all cached views. Then, upon demand, a view is simply retrieved from the cache. In this manner, database computation will be much more efficient and idle time will be minimized.

5.2.2 System-defined View Caching

Currently, the view cache gives the user a mechanism for defining important database views which should be updated incrementally after base table updates. By declaring that a view should be cached, the user is defining it to be a high priority view. Thus, the responsibility of defining high priority real-time views whose value will be needed often is left to the user.

This responsibility could be transferred to the database manager. The database system could be extended to maintain statistics on the frequency with which each query is made. The system would then decide which views should be maintained current for rapid retrieval. These frequently accessed views would then be cached and incremental updates would be used to maintain them in the face of database updates. In this manner, the time overhead added to updates due to view maintenance and the space overhead due to cached views will be minimized.

5.2.3 Generalized Caching and Incremental Updates

The ideas behind the cache and incremental update mechanism can be extended further. In Extended RTMS, only the results of JOIN and RETRIEVE operations are cached and updated incrementally. However, the idea of storing and incrementally updating a computed function value is a more general principle than this research indicates. Many functions can be transformed to store a computed value and then incrementally update this value based on an incremental change in the values (inputs) on which the function depends.

With computation intensive functions, incremental updates are much cheaper than complete recalculation of the function value. For example, large VLSI CAD simu-

lations need to be incremental so that small changes in inputs do not require complete regeneration of outputs.

A generalized caching mechanism has been implemented at Texas Instruments to store the results of arbitrary functions. Work has begun to integrate this mechanism with the Extended RTMS trigger mechanism for incremental result updates. When this work is completed, it will be possible to implement UNION, INTERSECTION, and the remainder of the RTMS relational operators incrementally.

5.2.4 Attach/Detach to Real-time Simulations

Many applications maintain their own data but need some database operations performed on the data. The RTMS:ATTACH operation allows a user-defined data structure of an implementation/storage-structure type recognized by RTMS to be grafted directly into an RTMS relation, without any recopying of data [11]. Once the data has been attached to a relation, all the power of the database, including relational operators and secondary indices, is available for use. After the data has been operated upon, the RTMS:DETACH operation is used to remove it from the database without destroying it.

If a real-time simulation were attached to an Extended RTMS database, the power of a real-time database system would be made available to this data. Real-time views of the simulation could be maintained, triggered upon, and displayed. With the use of existing business graphics routines, real-time graphs and charts could display the rapidly changing data in the simulation. Specialized routines would not have to be written to do these graphics.

In a circuit simulation, for example, a real-time bar chart could display the values of voltages at various nodes in the circuit. In a military battle simulation, a natural language query could be made to the database to yield an active map of the battle as in Figure 5.1.

The significance of this capability is that it permits simulations to be imple-

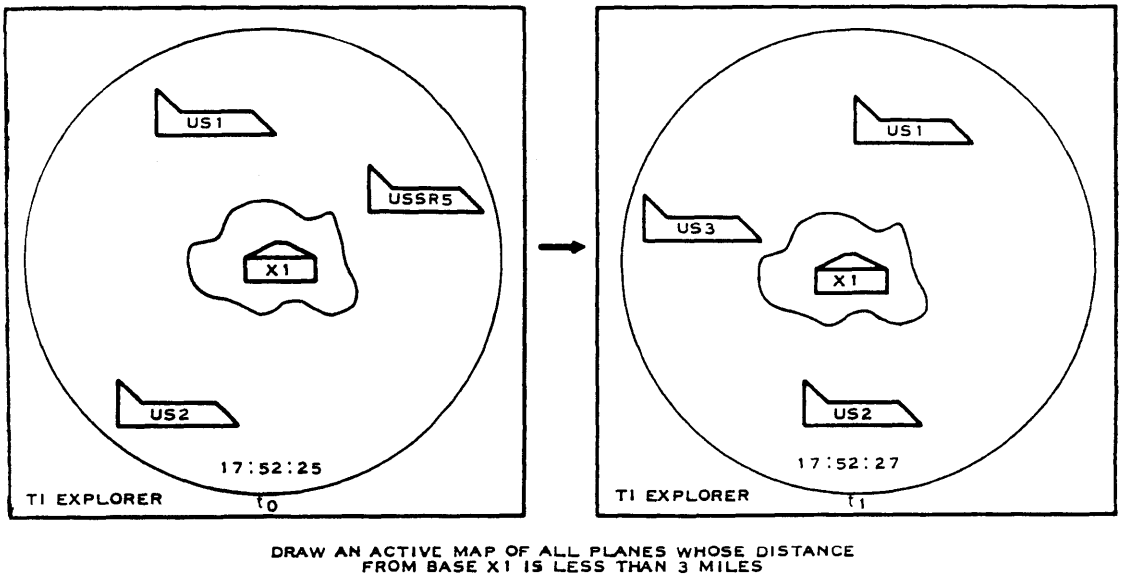

Figure 5.1: Extended RTMS Real-time Query Capability

mented independently, then combined with the database to give real-time windows on views of the simulation.

Each of these research directions is an exciting extension of the ideas developed in this thesis.

Bibliography

[1] *Release and Installation Information, Explorer RTMS Toolkit, Release 1.0.0.* Texas Instruments, Inc., Data Systems Group, Dallas, TX, 1985.

[2] Astrahan et. al. "System R: Relational Approach to Database Management." *ACM Transactions on Database Systems*, 1(2):106-110, June 1976.

[3] Blakeley, Larson, and Tompa. "Efficiently Updating Materialized Views." *Proceedings of SIGMOD '86*, pages 61-71, SIGMOD, 1986.

[4] Dennis L. Brandl and John McGehee. "Relational Data Base Extensions for Real-time Process Control." *Proceedings of Controls West Conference*, pages 218-229, 1985.

[5] Rajiv Enand and Craig Thompson. "Towards a Real-time Data Model." *Proceedings of Controls West Conference*, pages 230-234, 1985.

[6] Johnathan J. King. *Exploring the Use of Domain Knowledge for Query Processing Efficiency.* Technical Report HPP-79-30, Stanford Heuristic Programming Project, Dec 1979.

[7] Johnathan J. King. *Query Optimization by Semantic Reasoning.* PhD thesis, Stanford Department of Computer Science, May 1981.

[8] Bruce Lindsay et. al. "A Snapshot Differential Refresh Algorithm." *Proceedings of SIGMOD '86*, pages 53-60, SIGMOD, 1986.

[9] Cindy A. O'Reilly and Andrew S. Cromarty. "'Fast' is not 'Real-time': Designing Effective Real-time AI Systems. *Applications of Artificial Intelligence II*, pages 249-257, SPIE, 1985.

[10] M. Sloman and X. Andriopoulos. *Databases for Real-time Applications.* Technical Report DOC 83/9, Imperial College of Science and Technology, Dept. of Computing, Mar 1983.

[11] C. Thompson et. al. "RTMS: Toward Close Integration Between Database and Application." Dec 1986. Research Report.

[12] Jeffrey D. Ullman. *Principles of Database Systems.* Computer Science Press, Second Edition, 1982.