# RTSQL: EXTENDING THE SQL STANDARD TO SUPPORT REAL-TIME DATABASES

BY

JANET JAMIE PRICHARD

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN
APPLIED MATHEMATICAL SCIENCES

UNIVERSITY OF RHODE ISLAND
1995

DOCTOR OF PHILOSOPHY DISSERTATION

OF

JANET JAMIE PRICHARD

APPROVED:

Dissertation Committee

Major Professor

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

1995

# Abstract

This dissertation extends the standard query language SQL to support real-time databases to create RTSQL (Real-Time SQL). RTSQL includes extensions that specify: temporal consistency constraints on data, timing constraints on execution, bounds on use of system resources for predictability, and flexible transaction structure that relaxes traditional ACID transaction properties to better support real-time requirements.

To aid in identifying the necessary constructs that were added to the language, a model for real-time database systems was developed. The actual language constructs were developed based upon this model, and a subset of the constructs implemented as a prototype system. To evaluate the results, the implementation was tested for correct behavior, and the language constructs evaluated against requirements developed by a standards organization.

# Acknowledgments

It has been a long and winding road to the completion of this dissertation. At times, the road seemed to have no end, just a series of hills that had to be conquered one after the next. But through perseverance, and with the help of many good people around me, I finally made the finish line.

To start, I must thank my first advisor and friend Joan Peckham. She helped me make my dream of an academic career become a reality. She has served as a role model, and provided me with all of the encouragement and support a student could ask for.

Next, I thank my other advisor and friend, Victor Fay Wolfe. If it were not for his support, I seriously doubt I would have been able to finish this work. He helped me make the right connections, and provided me with the technical guidance I needed to finish my degree.

Thanks to Michael Anthony Squadrito (who has a crush on Bev), who helped me get my implementation off the ground. Lisa Cingiser DiPippo has been a good friend and great fellow researcher. And of course, there are all the folks in the RTSORAC research group who helped solidify ideas and provided useful input.

There are also a number of folks here at URI I would like to thank for their help and understanding over the years: Marge White, Lorraine Berube, Gerry Ladas, Frank Carrano, Norm Finizio, Choudary Hanumara, and Olga Verbeek.

Finally, I must thank my husband Michael, for all of his love and understanding during the last few months of this work. He is a good and loving husband and father, and I want him to know that his support has meant the world to me. And let me at last thank my children, Andrew Walter and Sarah Cathryn, for trying to understand that Mom had to work a lot to "finish her dissertation".

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Real-time databases are typically used to manage environmental data in computer control applications, such as programmed stock trading, medical patient monitoring, automated manufacturing, and military command and control. The environmental data is read from sensors and stored in the real-time database. For example, in a financial application for stock trading, a stock's price may be stored in the database, but only considered valid for 30 seconds from when the value was generated. If the value is not refreshed within that time period, it is said to be temporally inconsistent. Also, there may be information required from the database that will be required to determine whether a particular stock trade will generate a profit. This information must be retrieved within given deadlines so that decisions can be made on whether to actually initiate a stock trade. Thus, real-time databases are designed to support applications in which the freshness of data and the timeliness of processing are important.

This dissertation extends the standard query language SQL to support real-time databases to create RTSQL (Real-Time SQL). Query languages support the definition, manipulation, and control of data in a database system. For a real-time database, the query language must be extended to support the expression of timing constraints on both data and actions. To aid in identifying the necessary constructs that should be added to the language, a model for real-time database systems was developed. The actual language constructs were developed based upon this model, and a subset of the constructs implemented as a prototype system. To evaluate the results, the implementation was tested for correct behavior, and the language constructs evaluated against requirements developed by a standards organization.

## 1.1 Motivation

A real-time database system has three distinguishing features: the requirement of temporally consistent data, the requirement of timing constraints on executions, and the requirement that certain executions exhibit predictable timing behavior [Ram93, PDPW94, YWLS94]. These features are useful to time critical applications that need to collect, modify, and retrieve shared data. Support for these features will add new requirements to the database query language.

Data in a real-time database system must not only be logically consistent, but temporally consistent as well. This is due to the fact that data used by time critical applications must closely reflect the current state of the application environment. Data is collected at discreet intervals, and hence represents an approximation of reality. As time passes, this approximation becomes less accurate, until we reach a point at which the value is no longer reflective of the state of the environment. It is at this point in time that we say the data value is no longer temporally consistent.

Temporal consistency can be measured in two ways: *absolute temporal consistency* and *relative temporal consistency* [Ram93]. A piece of data is considered absolutely temporally consistent if and only if its age is within a specified time interval. For example, in a tracking system, the data corresponding to a contact, such as its speed, should be updated often (*e.g.* every five seconds). In this case, the value of the speed meets its absolute temporal consistency constraint as long as it is no more than five seconds old. A set of data items is relatively temporally consistent if and only if all of their ages are within a specified interval. Relative temporal consistency constraints are of interest when multiple data items are used in computations and all data items must represent the "same" snapshot of the environment. For example, if the tracking system computes the new location of a contact using the speed and bearing data items, it is important that the ages of these data items be relatively close (*e.g.* within two seconds).

The real-time database system should provide a means of detecting temporally inconsistent data. When the data is found to be temporally inconsistent, appropriate recovery actions can be taken. These actions may be specified as part of the database definition, or left to the user or application using the database system.

Actions in the database system are expressed using data manipulation statements. These statements are usually grouped together into units called *transactions*. Traditional transactions have been viewed as the means by which consistency and correctness of the database is ensured. They adhere to the *ACID* properties (**A**tomic, **C**onsistent, **I**solated, and **D**urable). Each of these properties is briefly described below:

| | | |
|---|---|---|
| Atomic | - | An atomic transaction implies the transaction runs in its entirety or not at all (all or nothing execution). |
| Consistent | - | Given an initial consistent monolithic database state a transaction must transform the database to a new consistent state. Consistency is defined by the expression of constraints on the entire database. |
| Isolated | - | Transactions are required to have the property of independent execution in conventional databases. Independence implies that transactions act in isolation from all other transactions and that there be no dependencies in execution among transactions. |
| Durable | - | The durability property of transactions implies that the results of a transaction are persistent and permanent. |

There are generally three types of transactions in a real-time database: *sensor* transactions, *update* transactions and *read-only* transactions [Ram93]. Sensor transactions are write-only transactions that obtain the state of the environment and write the sensed data to the database. Sensor transactions are typically periodic.

Update transactions can both read from and write to the database either periodically or aperiodically. Update transactions may be used to write values derived from computations or user input. Read-only transactions, such as some user queries, read data from the database and may also be either periodic or aperiodic.

Timing constraints on statements and transactions come from one of two sources (in this work, statements and transactions will be referred to as *actions* in the database). First, temporal consistency requirements of the data impose timing constraints on actions. For instance, the period of a sensor transaction is dictated by the temporal consistency constraints of the data item that it writes. Relative temporal consistency requirements may also force timing constraints on actions. If two data values are used together to derive a third value, the relative temporal consistency requirement on the pair may place a stricter constraint on their updates than the period imposed by the absolute temporal consistency of either one of the values.

The second source of timing constraints on actions is system requirements on reaction time. For instance, a user may query the database for information and request the result within a certain amount of time. Given the added dimension of time on actions, one of the interesting areas of study in real-time databases is that of transaction scheduling [BMHD89]. Not only must the schedules meet timing constraints, thus meeting the temporal consistency requirements, they must also maintain the logical (traditional) consistency of the data in the database.

Typically, there are two types timing constraints on actions: absolute timing constraints and periodic timing constraints. Absolute timing constraints are used to express earliest start times, latest start times, and latest finish times. These absolute timing constraints can be specified using expressions that involve absolute time (clock time) or relative time (an interval of time). Periodic timing constraints are used to express periodic executions, that can be based upon absolute time intervals of the data being updated.

RTSQL focuses on supporting *soft* real-time systems versus *hard* real-time systems. In hard real-time systems, violation of a timing constraint will have catastrophic consequences, so the system must guarantee that the timing constraints will be met. In a soft real-time system, violation of timing constraints will not have catastrophic consequences, but it is still not desirable. In these systems, some compensating action is usually performed when the timing constraint is violated. Soft real-time systems often recognize relative importance of transactions in the system, and utilize both deadlines and an importance level in determining transaction schedules.

The semantics of conventional transactions may also need to be redefined for real-time database systems. The inflexibility of the conventional ACID transaction properties often causes conservative execution of transactions that reduces real-time performance. For instance, a transaction that contains two sets of operations on two distinct portions of the database must wait to commit until all operations have been performed (as required by the atomicity property). This conventional transaction design prevents parts of the transaction from committing early and releasing data, even if there is no dependency on the rest of the transaction. The cost of this simple design is a loss in data availability and/or unnecessary transaction aborts. To better support real-time applications, researchers have investigated redefinition of various ACID requirements to allow for more flexibility and more data availability [YWLS94, FWP94]. These definitions utilize semantic information to determine to what degree the ACID properties must be enforced. The conventional database transaction processing model has no *a priori* knowledge about transaction execution or needs and can not therefore make *a priori* ordering decisions. Real-time transaction designers often have considerably more semantic information available. To incorporate this information, a richer set of transaction semantics that is defined by the database users (not by the database mangement system designers) must be specified. With increased semantic information available, the system can then make better decisions on execution sequences.

The following summarizes the requirements, beyond that of conventional database systems, of real-time database systems, :

- the absolute temporal consistency requirements of a data value
- relative temporal consistency requirements among a set of data values
- database partitions for enforcing consistency

- absolute timing constraints on statements and transactions
- periodic timing constraints on statements and transactions
- relative importance of transactions
- predictable timing behavior for certain situations
- relaxed ACID transaction properties.

Recently, there has been an effort to formalize the requirements of real-time database systems. Two real-time database standards study activities, the ANSI database systems study group (DBSSG) predictable real-time interface systems task group (PRISTG) and the U.S. Navy's next generation computer resources (NGCR) database management interface standards working group (DISWG) are studying real-time extensions [FS94, For94]. PRISTG focused on examining the current state of technology for real-time information management and determining the need for standardization in this area. They concluded that conventional databases lack support for real-time, and that the technology is mature enough to pursue standardization efforts.

The work of the DISWG committee was more detailed in that they actually generated a list of requirements for real-time databases (Appendix C). Their goal was to establish interface standards for use in mission-critical computing systems. Though the list of requirements is quite extensive, the expectation is that individual DBMS products would implement various subsets. A complete discussion of the requirements examined in this work appears in chapter 2.

The current SQL standard (SQL2) [MS92, Mel92] has no provisions for real-time database support. The standard does have mechanisms for constraint expression, support for expression of time, and rudimentary transaction structure – all of which provide a basis for developing real-time database extensions. Constraints in SQL provide a mechanism for specifying what constitutes correct data. For example, constraints can be used to specify an allowable range of values for an data item, or that the data value should not be left empty. In RTSQL, constraints are extended in two ways. First, constraints on data are extended to specify the interval of time that a data item is considered valid. Second, data manipulation statements and transactions can have timing constraints specified upon their execution. We also introduce *directives*, which are added language features that express architectural dependencies and resource limitations that are necessary for achieving predictable timing behavior.

## 1.2 Research Goals

The goal of this work is to develop a set of language constructs which could be used as a basis for creating a standard query language for real-time database systems. This development includes specification of the language constructs, evaluating the specification for completeness against published requirements, and implementing some of the constructs to evaluate their feasibility.

## 1.3 Our Approach

The DISWG requirements served as a basis for determining our extensions to SQL for supporting real-time database systems. This lead to the development of a model for real-time databases called RTSORAC (Real-Time Semantic Objects, Relationships, and Constraints). This model was developed to provide a conceptual basis for exploring the issues related to real-time databases and an abstract foundation on which the constructs of RTSQL are based. In developing RTSQL, efforts were made to adhere to the philosophy of SQL as a declarative language. In a declarative language, you specify *what* you want done, not *how* it is to be done. For example, SQL provides a statement for retrieving information from the database called a SELECT statement. Using this statement, you would retrieve the names of the students with a grade point average greater than 3.0 as follows:

```
SELECT name FROM students WHERE gpa > 3.0
```

Note that the statement does not specify how the information is to be found, that is left to the underlying system to determine.

Once the language constructs were developed, a subset of constructs was implemented to produce a prototype system. This implementation was then tested and evaluated. The constructs of RTSQL were also evaluated against requirements developed by the NGCR DISWG committee.

## 1.4 Outline of Dissertation

In Chapter 2 we review related work, including a discussion of the DISWG requirements, the SQL standard, and real-time languages which influenced this work. Chapter 3 defines our RTSORAC model for real-time databases. Chapter 4 describes the language constructs of RTSQL. Chapter 5 describes the implementation of the prototype system. Chapter 6 discusses the testing of the implementation, and examines RTSQL against the DISWG requirements. Chapter 7 discusses our current efforts with the ANSI working groups, presents the contributions and limitations of our work, and discusses future work.

# Chapter 2

# Related Work

This chapter describes some of the related work that has been done in the areas of real-time database requirements and query languages. The first section will provide an overview of the real-time database requirements developed by DISWG. The next section will discuss the SQL standard and extensions to the standard currently under consideration. This section will also discuss some of the features of real-time programming languages, since many of the extensions in RTSQL were influenced by these languages.

## 2.1 Real-time Database Requirements

The DISWG requirements for real-time database systems have been published in "Requirements for Military Database Management Systems"[Gor93]. The requirements fall into nine classes which are documented in [Gor93] as follows: (Note: MCCR - Mission-critical computer resources)

> *(1) General requirements. This class specifies general goals (e.g., scalability, modularity, extensibility, configurability) of interface standards.*
>
> *(2) Basic database management services. This class specifies the basic services typically provided by today's general-purpose DBMSs and which must be included in NGCR DBMS interface standards.*
>
> *(3) Distribution. This class addresses the distribution of data across homogeneous, tightly-coupled databases, which together form a single logical database*

*known as a distributed database. It specifies requirements for distributed database management system interfaces.*

*(4) Heterogeneity. This class addresses the distribution of data across heterogeneous, autonomous databases. It specifies capabilities for remote database access, global transactions, multidatabase systems, and federated database systems.*

*(5) Real-time processing. This class addresses the needs of MCCR DBMSs for hard, firm, and soft real-time processing. It specifies capabilities for managing the updating of time-constrained data and the execution of time- constrained transactions. The capabilities enable users and/or application programs to control resource usage in accordance with mission needs.*

*(6) Fault tolerance. This class addresses the needs of MCCR DBMSs for reliability, availability, fault tolerance, and graceful degradation. It specifies capabilities for managing the collection of fault information and the formulation of fault responses.*

*(7) Integrity. This class addresses the needs of MCCR DBMSs for data integrity. It specifies several integrity-preserving mechanisms: domains, keys, referential integrity constraints, assertions, triggers, and alerters.*

*(8) Security. This class addresses the needs of MCCR DBMSs for multilevel security. It specifies requirements for multilevel secure DBMSs, including labeling, mandatory and discretionary access control, identification and authentication, and auditing.*

*(9) Advanced database management services. This class specifies some of the functionality that is typically associated with object-oriented database management systems and knowledge base management systems. Such advanced functionality is required for the management of complex data and rules.*

These requirements are quite extensive, and as such, the expectation is that no single DBMS will be able to provide support for all of the requirements. Also, given the breadth of topics within these requirements (such as distribution and heterogeneity), the expectation is that a suite of standards may be necessary to fulfill all of the requirements.

Of particular interest in this work are real-time processing class and the fault tolerance class. The requirements in both of these classes is discussed in the following sections.

### 2.1.1 Real-time Processing Requirements

Many of the extensions developed for RTSQL will address the requirements in this class. The following paragraphs will provide the brief description of each of these requirements as it appears in [Gor93] (including requirements numbers). A paragraph which further clarifies the requirement may follow as needed.

*3.5.2.1* ***Modes of real-time.*** *The NGCR DBMS interface standards shall provide support for hard real-time, firm real-time, and soft real-time modes of operation.*

Three levels of real-time are identified: hard real-time, firm real-time, and soft real-time. Recall from section 1.1 that in hard real-time systems, constraint violations will have catastrophic consequences, and in soft real-time systems, constraint violations will not have catastrophic consequences, but are is still not desirable. Firm real-time is similar to soft real-time in that constraint violations will not cause catastrophic consequences. But, in soft real-time systems, the results may still have some value, even though the constraint is violated. For example, partial results might be utilized to extrapolate a final result if the timing constraint is violated before all of the data can be retrieved. In firm real-time, the results are usually of no value if the constraint is violated.

*3.5.2.2* ***Real-time transactions.*** *The NGCR DBMS interface standards shall provide the capability for users to issue real-time transactions where ACID properties (such as the isolation property, which can be relaxed via the specification of alternative concurrency control correctness criteria) are applied selectively, and where start events, deadlines, periods, and criticality of the real-time transactions are specified. (See Requirements 3.5.2.3 and 3.5.2.5.)*

The start event specifies an interval of time when a transaction may begin execution. The deadline specifies the latest possible time the transaction must complete execution. Periodic execution allows a transaction to run at regular intervals, for example, every 30 seconds. Specification of the criticality of a transaction provides the underlying system with the relative importance of transactions to the system.

*3.5.2.3* ***Concurrency control correctness criteria.*** *The NGCR DBMS interface standards shall provide the capability to specify concurrency control correctness criteria. (See Requirements 3.5.2.5 and 3.5.2.6.)*

This criteria is used to determine how concurrent access to shared data by transactions should be handled. For example, one criteria might be to not allow any concurrent access to the data (mutual exclusion). Another might be to allow concurrent reads of the data, but writes must be done exclusively (read/write locking)[BHG87]. A thorough overview of concurrency control techniques can be found in [DiP95].

*3.5.2.4* ***Temporal consistency.*** *The NGCR DBMS interface standards shall provide the capability for users to specify data temporal consistency constraints. (See Requirements 3.5.2.5 and 3.5.2.7).*

This requirement was discussed in section 1.1. It allows users to specify an interval of time in which the data is considered temporally valid.

*3.5.2.5* ***Real-time scheduling.*** *The NGCR DBMS interface standards shall provide DBMS real-time scheduling that attempts to maximize meeting timing constraints and criticality (the synthesis of these two requirements is left undefined here) of transactions, as well as attempting to maintain both logical and temporal consistency of data. The NGCR DBMS interface standards shall require that real-time scheduling support analysis of predictable timing behavior (e.g., by bounding priority inversion).*

This requirement brings up two important ideas in real-time databases. First, is the fact the a real-time database often has to strike a balance between the logical consistency of data and the temporal consistency of data[Ram93, Ulu92, Son90]. Traditional database systems focus on maintaining logical consistency of data, and schedule transactions without concern for when the transaction will run, or how long it will take to execute. In real-time databases, actions may have timing constraints which must be met to maintain the temporal consistency of data. In some situations, these timing constraints can only be met if the data logical consistency is sacrificed. Or it may be that the logical consistency can only be maintained if timing constraints are violated, leaving data temporally inconsistent. Since these scenarios are often present in real-time database systems, a compromise is established through the concept of imprecision.

There are two forms of imprecision that can be associated with a data value. The first is logical imprecision, where the value of a data item has a delta value associated with it. If the logical consistency of the data item is sacrificed, then this delta value is used to quantify the amount that the data value may differ from what it would have been without the sacrifice. The second is temporal imprecision, where the timestamp value associated with a data item also has a delta value associated with it. In this case, the delta value is used to quantify the amount of time that has elapsed since the value was properly updated. This could occur in situations where temporal consistency is sacrificed.

The second major idea is predictability[Ram93, YWLS94]. Predictabilty is most often discussed in the context of timing constraints on actions and transactions. If the system is provided with enough information, it may be able to predict whether or not an action or transaction will be able to complete by its given deadline. This information could include defining or computing the maximum amount of time it takes to perform a certain action (worst case execution time). Computation of this value is possible in situations where lower level operations have been bounded through resource limits. For example, if a table of information has a maximum size, the time it takes to search the table can be bounded.

*3.5.2.6* ***Bounded logical imprecision.*** *The NGCR DBMS interface standards shall allow logical imprecision of data; it shall provide the capability to constrain these imprecisions.*

As mentioned in requirement 3.5.2.5, a logical imprecision of data may occur in an attempt to meet timing constraints on actions. This requirement states that this value should be quantified and that there should be some mechanism for bounding this value.

*3.5.2.7* ***Bounded temporal imprecision.*** *The NGCR DBMS interface standards shall allow temporal imprecision of data; it shall provide the capability to constrain* *these imprecisions.*

As mentioned in requirement 3.5.2.5, a temporal imprecision of data may occur in an attempt to maintain logical consistency of the data. This requirement states that this value should be quantified and that there should be some mechanism for bounding this value.

*3.5.2.8* ***Main memory data.*** *The NGCR DBMS interface standards shall provide the capability to specify that certain parts of the database should be maintained exclusively in main memory. The NGCR DBMS interface standards shall require that the DBMS still be responsible for maintaining persistence of this main memory data.*

This requirement is intended to provide more predictable access times to data. In a traditional database system, information is usually stored on secondary storage devices such as disks. Getting information to and from such devices is often unpredictable [Ram93]. If the information can be stored in main memory, this eliminates the problem of unpredictable access time associated with secondary storage access.

*3.5.2.9* ***Time fault tolerance.*** *The NGCR DBMS interface standards shall support time fault tolerance. That is, violations of transaction timing constraints and data temporal consistency constraints are faults and shall be treated as such by the fault-tolerance capabilities of the standard, as specified in Section 3.6.*

This requirement is addressed in more depth in the next section. It simply states that if timing constraints on data or actions are violated, the system should be able to detect these violations.

*3.5.2.10* ***Resource utilization limits.*** *The NGCR DBMS interface standards shall allow the specification of worst-case resource utilization limits (at least, CPU time, memory, devices, and data objects) for transactions. Violations of these limits are faults and shall be treated as such by the fault-tolerance capabilities of the standard, as specified in Section 3.6.*

*3.5.2.11* ***Compilable DML.*** *The NGCR DBMS interface standards shall provide a compilable DML that yields a minimal run-time burden.*

In real-time systems, many of the transactions are known apriori. This requirement recognizes that fact, and provides for these transactions to be compiled into a form more readily executed by the database system.

### 2.1.2 Fault Tolerance Requirements

Real-time databases are often used in systems where the database must be reliable. It must be able to continue execution, possibly with degraded functionality or performance, despite failure of some of its constituent components[Gor93]. These failures are considered to be faults in the database system. Some of these requirements in this section are beyond the scope of RTSQL, but have been left here for completeness. As in the previous section, the requirements of interest will be followed by a paragraph which further clarifies the requirement as needed.

*3.6.2.1* ***Collection of fault information.*** *The NGCR DBMS interface standards shall specify the fault information (e.g., the component that failed, the number of times the fault occurred, when the faults occurred) to be collected. The standard shall also specify a minimal set of faults for which the specified information shall be collected. This set shall include, but is not limited to, the following faults:*

- *Database constraint violations (e.g., range constraints, referential integrity constraints, temporal consistency constraints).*
- *Transaction timing faults.*
- *Transaction resource utilization violations.*

This requirement is somewhat self explanatory. It simply states that the system should not only recognize that a fault has occurred, but should maintain information related to the fault. This is already a common practice of most commercially available database systems. The primary difference here is that this should be extended to cover constraint violations for time constrained data and actions, as well as possible resource limit violations.

*3.6.2.2* ***Retrieval of fault information.*** *The NGCR DBMS interface standards shall provide for the retrieval of DBMS fault information.*

Not only should the database system maintain the information related to the fault, but provide some mechanism for retrieving it. Again, this is a feature already found in most commercially available database systems.

*3.6.2.3* ***Initiation of diagnostic tests.*** *The NGCR DBMS interface standards shall provide for the initiation of DBMS diagnostic tests.*

This requirement is beyond the scope of RTSQL.

*3.6.2.4* ***Retrieval of results of diagnostic tests.*** *The NGCR DBMS interface standards shall provide for the retrieval of the results of DBMS diagnostic tests.*

This requirement is beyond the scope of RTSQL.

*3.6.2.5* ***Operational status.*** *The NGCR DBMS interface standards shall provide access to the operational status of DBMS components.*

This requirement is beyond the scope of RTSQL.

*3.6.2.6* ***Fault detection thresholds.*** *The NGCR DBMS interface standards shall provide for the specification of fault detection thresholds, which shall include, but not be limited to, the number of faults that if detected within a certain amount of time is treated as a failure (e.g., the number of retry attempts of aborted transactions before a failure of that transaction is reported).*

Most database systems simply recognize that a fault has occurred, they provide no mechanism for differentiating the first occurrence from a number of occurrences. In real-time systems, the frequency of some operations in the database (e.g. every millisecond) make this an unreasonable assumption.

*3.6.2.7* ***Specification of fault responses.*** *The NGCR DBMS interface standards shall provide for the specification of actions to be taken at the occurrence of a fault. They shall support at least the following actions:*

- *Restart of a specified set of transactions at a database's specified past state or with only a specified part of the database replaced by its past state.*
- *Rollback of specified transactions that have started, but not yet committed, so that their effects are not realized in the database.*
- *Use of specified backup components as primary components (e.g. other versions of the database).*
- *Providing notification of a fault to a specified set of DBMS components to allow them to initiate recovery.*
- *Providing notification of a fault to a specified location outside of the DBMS.*
- *Reconfiguration of DBMS components (see next requirement).*

*Furthermore, the NGCR DBMS interface standards shall allow for each of these actions to be applied selectively. Also, these actions may fall under time-constrained execution described in the "Real-Time Processing" section.*

The first two actions, "restart of a specified set of transactions" and "rollback of specified transactions" are directly related to the commit and abort semantics of transactions - the idea of all or nothing execution of the actions of a transaction. Traditional database systems usually provide a recovery mechanism which maintains this view of a transaction.

The third point, "use of specified backup components", addressed component failure, an issue beyond the scope of RTSQL.

The fourth point, "providing notification" is one of the most important aspects of fault tolerance. This point addresses the need for the system to support some mechanism for the user to specify a recovery action which should be taken if a fault occurs.

The fifth and sixth points are beyond the scope of RTSQL.

*3.6.2.8* ***Reconfiguration.*** *The NGCR DBMS interface standards shall support dynamic reconfiguration of the DBMS components based on reconfiguration of the underlying operating system and hardware. Reconfiguration includes, but is not limited to, enabling/disabling components, adding/deleting components as members of specified groups and reassigning resources to components. Reconfiguration must be allowed as a response to a fault, as in the previous requirements, or at the discretion of certain DBMS components.*

This requirement is beyond the scope of RTSQL.

*3.6.2.9* ***Replicated components.*** *The NGCR DBMS interface standards shall not preclude the use of replicated components.*

This requirement is beyond the scope of RTSQL.

## 2.2 Language Support for Real-Time Databases

Though a number of database languages exist, this work will be based upon the ANSI/ISO standard database language SQL. SQL was specifically chosen because it represents an effort by the database community to provide a common language for accessing a variety of database systems. The specification of SQL[Mel92] also clearly defines the semantics of the statements which constitute the language.

This section will provide an overview of SQL. It will also discuss other major extensions to the language which currently exist. This is then followed by a discussion of common features of real-time languages, in particular RTC[WDL93], which has influenced the development of RTSQL.

### 2.2.1 SQL2 - The Current Database Standard

Standardization efforts for database management systems are relatively new [Gal91]. The American National Standards Institute (ANSI) and the International Standards Organization (ISO) standardization groups first approved the database language standard SQL in 1986. A revised version, called SQL2, was approved in 1992.

Currently, ANSI has working groups determining the next revisions to SQL, often referred to as SQL3. Some of the features under development in SQL3 include support for abstract data types and objects, triggers, and condition handling (including exception handling). Other efforts include support for persistent stored modules (SQL/PSM) [Mel95] and remote data access (SQL/RDA) [RDA]. There have also been a number of efforts to create extensions to the standard for various special purposes including: security [ST90], temporal data [Sno94], and multi-media data [Gal92].

The ANSI charter is to serve as an independent, voluntary institution for the development, management, and coordination of national standards. Within ANSI is the Computers and Information Processing Committee (X3). X3 is responsible for the development and maintenance of the standard database language SQL. Organizations can join the committee and send any number of representatives to the meetings, which are held six times a year. Each organization is allowed to have one representative that is known as the voting member. This individual represents the interests of the organization. Many of the major vendors of commercially available database systems participate in this committee including Sybase, IBM, Oracle, Microsoft, and Borland just to mention a few. Other members of the committee include the National Institute for Standards (NIST), and many branches of the military, including the Army and the Navy. Note that there are few academic participants in this process. This is primarily due to the cost involved with membership, and the attendance requirements which incur high travel costs. Also note that each organization can have only one voting member, this is to keep any one organization from dictating what the standard will contain. But, interestingly enough, any small company or organization which joins the committee has the same voting power as a large corporation such as Microsoft.

The current SQL standard (SQL2) [MS92, Mel92] has some provisions that can provide the basis for real-time database support. The standard provides mechanisms for constraint expression, support for expression of time, and rudimentary transaction structure. Although these features provide a basis for developing real-time database extensions, further extensions are necessary. These include extending the notion of constraints to time, providing mechanisms to specify low level information to the database system, and defining a more flexible transaction model.

SQL2 supports the definition, manipulation, and control of data in a relational database system. It based on the model of a single, monolithic database comprised of a collection of relational tables. Statements in SQL2 are categorized as data manipulation statements, data definition statements, and database management statements [MS93]. Data definition statements are used to define and/or modify the structure of the database. For example, data definition statements are used to define the structure of the tables in the database. Data manipulation statements are used to retrieve, store, remove, and update data stored in the tables in the database. Management statements are used to specify parameters that can affect the execution of other statements. For example, management statements can be used to control access to portions of the database based upon user identifications.

The following paragraphs highlight some of the features of SQL2. A more detailed description of condition handling in SQL2/PSM is also provided since condition handling is a feature that is important in RTSQL.

**Data Types.** SQL2 provides a limited number of data types to the end user. It also provides a `CREATE DOMAIN` statement that allows the user to associate a more meaningful name with one of the provided data types. Real-time applications such as weather forecasting, medical monitoring, and defense systems need a variety of non-standard data types specific to their applications domain. But current efforts within SQL3 and SQL/MM are already addressing representation of more complex data types. SQL3 provides BLOB (Binary Large OBjects) and CLOB (Character Large OBject) for large unstructured data. SQL3 also supports the definition of objects that have attributes and methods. SQL/MM efforts are currently focusing on full text data and spatial data. Thus, RTSQL will not address complex data type definitions directly.

**Constraints.** Constraints in SQL2 are mechanisms for specifying the logical consistency requirements of data. They can be specified on columns, tables, and as stand alone entities (called assertions) within the database. For example, a constraint can be used to specify a range of values for a data item, or to make sure that a foreign key in one table corresponds to a primary key in another.

**Condition Handling.** One area currently being addressed by the SQL2/PSM project[Mel95] is support for condition handling. Condition handling allows a more active response to the completion of an SQL statement. When a statement is executed, it will either raise an *exception* condition or a *completion* condition. A number of system-defined exception and completion conditions are provided through a status parameter called `SQLSTATE`. This parameter is a character string which contains codes representing the completion status of the SQL statement. SQL/PSM allows a condition handler to be associated with these conditions, or with user-defined exception conditions.

The scope of a condition handler in SQL2/PSM is specified by using a compound statement. This compound statement may contain one or more statements, condition handler declarations, and local variable declarations. Thus, a declaration of a condition handler will always appear in the context of a compound statement. If a single statement is to have its own condition handlers, then a compound statement containing that single statement must be created. A condition handler is instantiated when the compound statement containing it is executed. The handler is destroyed when the compound statement completes execution. Thus, when a condition is raised, if there is a corresponding handler in the set of instantiated condition handlers, it is executed. If an exception condition occurs during the execution of a statement, and there is no corresponding condition handler, it becomes an "unhandled exception" and execution of the current statement is halted. If a completion condition occurs when a statement is executed, and no condition handler can be located, the condition is ignored, and execution continues at the statement following the one raising the completion condition.

Two models of condition handling are supported, the termination model and the resumption model. The first model, the termination model, is primarily intended to be used with exception conditions. In this model, when the outcome of a statement is known, control is passed to the appropriate condition handler (if it exists), and execution of the corresponding frame is abandoned. Upon completion of the condition handler, control is returned to the same point at which the original frame was to return. This model is supported by using the handler types `EXIT` and `UNDO`. A handler of type `EXIT` behaves as stated above, whereas a handler of type `UNDO` will do a rollback of all the changes in the corresponding frame before executing the `<handler action>`. In both cases, the remainder of the original frame is abandoned.

The second model is the resumption model. This model allows the frame to resume execution after the condition is handled. There are two types of handlers corresponding to this model, `CONTINUE` and `REDO`. A handler of type `CONTINUE`, after executing the `<handler action>`, will return control to the statement following the one that caused the handler to be invoked. A handler of type `REDO` will cause a rollback of the changes in the corresponding frame, execute the `<handler action>`, then return control to the first statement of the original frame.

**Time Specification.** SQL2 provides sufficient syntax and semantics for specification of timing expressions. There are three *datetime* data types: `DATE`, `TIME`, and `TIMESTAMP`. These data types can be used to express absolute time, such as 9am. There is also an *interval* data type called `INTERVAL`, that can be used to express a period of time, such as 5 minutes. SQL also supports three *datetime* valued functions: `CURRENT_DATE` returns the current date, `CURRENT_TIME` returns the current time, and `CURRENT_TIMESTAMP` returns the current date concatenated with the current time. The arithmetic operators `+`, `-`, `*`, and `/`, and the usual comparison operators (`=`, `<>`, `<`, `<=`, `>`, `>=`) have been defined over *datetime* data types and *interval* data types. TSQL2 [Sno94] has proposed a precise definition on representation of time within the database. This definition includes the concept that time has a discrete representation, and that the smallest unit of time is called a chronon.

**Transactions.** SQL2 has only very basic capabilities for specifying the notion of a transaction. A transaction is a sequence of SQL statements that adhere to the *ACID* properties [BHG87, OV91]. The *ACID* properties are: Atomicity, Consistency, Isolation and Durability of transaction execution. An atomic transaction requires all or nothing execution. Consistency requires that transactions given an initial consistent database state transform the database to a new consistent state. Consistency is determined by the evaluation of constraints defined on the (entire) database. Isolation requires that a transaction execute in virtual isolation from all other transactions and that there be no dependencies in execution between transactions. Durability requires that the results of a transaction are persistent and permanent.

### 2.2.2 Current Extensions to SQL

A number of specialized areas are currently being addressed by the standards committee. A brief description of each of these ares is provided below.

**Temporal Query Languages.** One important use of database systems is the ability to archive information. These database systems, known as temporal database systems, provide mechanisms for maintaining the time interval that a set of data was valid. For example, using an employee database, you might want to determine an employee's salary in May of 1990 and June of 1993. Presumably the employee's salary will be different on those two dates from what it is now. The fact that the database system maintains this salary history is a characteristic of temporal databases.

Note that this idea of data and time intervals differs from that considered in real-time database systems. In a temporal database, the time period that a set of data is valid is an integral part of the information. In a real-time database, the interval is viewed as a constraint on the data.

**Multi-media.** Traditional database systems offer a limited set of data types to the end user. Within the last few years there has been a proliferation of more complex data such as sound, pictures, and animation available in a digitized form which can be stored on a computer. To address the need to store this information within a database system a subgroup was formed by the standards committee to examine these complex data types.

### 2.2.3 Real-Time Languages

Real-time programming languages usually provide some mechanism of expressing timing constraints and execution constraints on operations. In a database system, the data manipulation language is the portion of the query language which provides the operations for retrieving and modifying the data stored in the system. In a real-time database system, the data manipulation language also needs to provide primitives for expressing timing constraints and execution constraints on operations and transactions. Thus, many of the features found in real-time languages could be incorporated into a data manipulation language for a RTDBS. A brief summary of these features follows in the paragraphs below.

**Timing Constraints.** Timing constraints are used to express the start times and deadlines (or completion times) of operations. These constraints may specify an interval of time in which some action must start or complete execution. Timing constraints may also be used to specify periodic execution of an action. Some languages, such as Ada, only provide only very primitive timing constraints such as a delay clause. Others, such as RTC [WDL93] provide a more complete set of primitives including *after* and *before* clauses that can be used to specify the earliest starting time and the latest starting time respectively for an action. RTC also provides clauses for the completion time and maximum execution time of an action. In FLEX [LN88], limits on time and resources are specified in a constraint block. These limits include the maximum duration of the block, and the earliest start time and finish time for the block. In FLEX, the start and finish times can be based upon start and finish times of other blocks. Real-Time Euclid [KS86] requires every program to express its timing constraints, and forbids constructs (such as recursion) that take arbitrarily long to execute.

**Execution Constraints.** Execution constraints are used to express synchronous and asynchronous operation execution and the relative order of the operation executions (e.g. precedence, simultanaity). For example, RTC provides a clause for specifying that a set of actions be executed simultaneously. Ada provides a rendevous mechanism which allows tasks to synchronize their execution with other tasks in predetermined ways. Other languages, such as C, provide easy access to operating system primitives that can be used to generate signals and can be used to wait until a signal is posted. These primitives can be used to establish simple synchronization. In Real-Time Euclid, a signal and a wait primitive are provided where the wait primitive has been extended to specify a time bound.

**Exception Handling.** Most real-time languages provide a set of language constructs for expressing constraints. They also usually provide a mechanism for recovery should a constraint be violated (exception handling). The underlying run-time system must provide mechanisms which can be accessed from the programming language to determine if a constraint has been violated. Many languages, such as Ada and C++, provide an exception handling mechanism that allows a user to react to user or system defined exception conditions. For example, an exception handler could be specified to handle a divide by zero error. Real-Time Euclid, Flex, and RTC all provide exception handling for timing constraint violations. Real-Time Euclid also provides time-bounded exception handlers.

**Summary.** Many of the timing constraints specified on actions in RTSQL have a basis in real-time programming languages (RTC, Flex, Real-Time Euclid). Also, the ability to react to violations of these constraints (which can be specified in RTSQL) is another feature common to real-time programming languages. This work in the area of real-time languages provides a strong basis for many of the timing constraint constructs in RTSQL.

# Chapter 3

# The RTSORAC Model

To aid in the development of the language constructs of RTSQL, a conceptual model called RTSORAC was developed for real-time databases. The RTSORAC model is loosely based upon the ER (Entity Relationship) model[Che76]. The ER model provides mechanisms not only for modeling data, but relationships between data as well. One of the strengths of the ER model is its ability to maintain semantic information about the data. The RTSORAC model provides these capabilities, and extends them for real-time. It also has an additional component for modeling transactions.

Though the RTSORAC model was originally designed to support real-time object-oriented databases, it is general enough to be used as a conceptual model for other database architectures, such as relational databases. The RTSORAC model provides a foundation for many of the language constructs that appear in RTSQL, especially in the areas of constraint specification. RTSORAC also provides the foundation for the transaction specifications in RTSQL. This chapter will first provide a detailed description of the RTSORAC model.

RTSORAC has three components that model the properties of a real-time database: *objects*, *relationships* and *transactions*. *Objects* represent data-base entities. They are used to specify the structure of the data and to specify constraints on the data. *Relationships* represent associations among the database objects. They are used to specify associations between *objects* and to specify constraints on groups of objects participating in the relationship. *Transactions* are executable entities that

$$\begin{array}{rcl}
\textbf{Object} & = & \langle N, A, M, C, CF \rangle \\
N & = & UniqueID \\
A & = & \{a_1, a_2, ..., a_m\} \text{ where attribute } a_i = \langle N_a, V, T, I \rangle \\
M & = & \{m_1, m_2, ..., m_n\} \text{ where method } m_i = \langle N_m, Arg, Exc, Op, OC \rangle \\
C & = & \{c_1, c_2, ..., c_s\} \text{ where constraint } c_i = \langle N_c, AttrSet, Pred, ER \rangle \\
CF & = & \text{compatibility function}
\end{array}$$

Figure 3.1: Object Characteristics in RTSORAC

access the objects and relationships in the database. The chapter will conclude with a discussion of the features of the RTSORAC model which will appear in RTSQL.

## 3.1 Objects

An *object* (Figure 3.1) consists of five components, $\langle N, A, M, C, CF \rangle$, where $N$ is a unique name or identifier, $A$ is a set of attributes, $M$ is a set of methods, $C$ is a set of constraints, and $CF$ is a compatibility function. Attributes, methods, constraints, and the compatibility function are described below. Figure 3.2 illustrates an example of a **Train** object (adapted from [Boo91]) for storing information about a railroad engine in a database.

### 3.1.1 Attributes

$A$ is set of attributes for the object, where each attribute is characterized by $\langle N_a, V, T, I \rangle$. $N_a$ is the name of the attribute. The second field, $V$, is used to store the value of the attribute, and may be of some complex data type. The next field, $T$ is used to store the timestamp of the attribute, and is of some data type capable of expressing a time. Access to the timestamp of an attribute is necessary for determining temporal consistency of the attribute. For example, in the **Train** object, there is an attribute for storing the oil pressure called `OilPressure` to which a sensor regularly provides readings. This update is expected every thirty seconds, thus the `OilPressure` attribute is considered temporally inconsistent if the update does not occur within that time frame. The timestamp value of the `OilPressure`

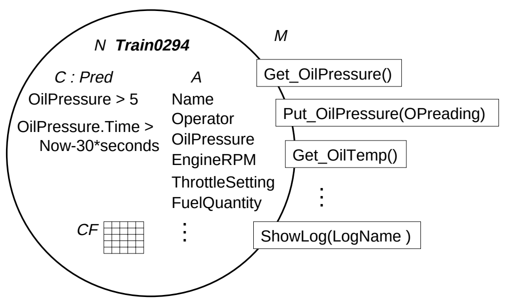


Figure 3.2: Example of **Train** Object

attribute must be utilized by the system to determine that the update did not occur as expected.

The last field $I$ is used to store the amount of logical imprecision associated with the attribute, and is of the same type as the value field $V$. In order to meet real-time constraints it may not be possible to maintain precise data values. Furthermore, many real-time control applications allow a certain amount of imprecision. For instance, within the **Train** object, the value of `OilPressure` attribute may not have to be precise.

### 3.1.2 Methods

The third component of an object, $M$, is a set of methods, where each method is of the form $\langle N_m, Arg, Exc, Op, OC\rangle$. $N_m$ is the name of the method. $Arg$ is a set of arguments for the method, where each argument has the same components as an attribute, and is used to pass information in and/or out of the method. $Exc$ is a set of exceptions that may be raised by the method to signal that the method has terminated abnormally.$Op$ is a set of operations which represent the implementation of the method. These operations include statements for conditional branching, looping, I/O, and reads and writes to an attribute's value, time, and imprecision fields.

The last characteristic of a method, $OC$, is a set of operation constraints. An operation constraint is of the form $\langle N_{oc}, OpSet, Pred, ER\rangle$ where $N_{oc}$ is the name of the operation constraint, $OpSet$ is a subset of the operations in $Op$, $Pred$ is a Boolean expression, and $ER$ is an enforcement rule. The predicate is specified over *OpSet* to express precedence constraints, execution constraints, and timing constraints [WDL93]. The enforcement rule is used to express the action to take if the predicate evaluates to false. A more complete description of an enforcement rule can be found in the next section on constraints.

Here is an example of an operation constraint predicate in the **Train** object:

```
Pred :    complete(Put_OilPressure) < NOW + 5*seconds
```

A deadline of `NOW + 5*seconds` has been specified for the completion of the `Put_OilPressure` method. Note the use of a special atom `complete(e)`, which represents the completion time of the executable entity `e`. Other atoms that are useful in the expression of timing constraints include `start(e)`, `wcet(e)`, and `request(e)` which represent the execution start time, worst case execution time, and the execution request time of entity `e` respectively.

### 3.1.3 Constraints

The fourth component of an object is a set of constraints, $C$, which permits the specification of correct object state. Each constraint is of the form $\langle N_c, AttrSet, Pred, ER\rangle$. $N_c$ is the name of the constraint. *AttrSet* is a subset of attributes of the object. *Pred* is a Boolean expression that is specified using attributes from the *AttrSet*. The predicate can be used to express the logical and temporal consistency requirements of the data stored in the object by referring to the value, time, and imprecision fields of the attributes in the set.

The enforcement rule ($ER$) is executed when the predicate evaluates to false, and is of the form $\langle Exc, Op, OC\rangle$. As with methods, *Exc* is a set of exceptions that the enforcement rule may signal, *Op* is a set of operations that represent the implementation of the enforcement rule, and *OC* is a set of operation constraints on the execution of the enforcement rule.

Logical and temporal consistency constraints on data require two distinct methodologies for evaluation. Predicates based upon logical consistency requirements are evaluated when write operations are performed on the attributes in *AttrSet*. All writes in the database are the result of a transaction which may be either user initiated or system initiated. Hence, an enforcement rule associated with such a predicate will always be executed in the context of a transaction. This execution may be synchronous or asynchronous and may involve signaling an exception that is propagated back to the transaction. Predicates based upon temporal consistency requirements may be violated simply due to the passage of time and the semantics of predicate evaluation can vary. Once a constraint violation has been detected, the corresponding enforcement rule is executed. It is possible that there is no context (such as a transaction) for the execution of the enforcement rule. In this case the implementation must provide a means of handling exceptions raised outside of the context of a transaction, perhaps through the use of a monitor that can detect and act upon signaled exceptions.

For example, as mentioned earlier, the **Train** object has an oil pressure attribute that is updated with the latest sensor reading every thirty seconds. To maintain the temporal consistency of this attribute, the following constraint is defined:

```
N :         OilPressure_avi
AttrSet :  {OilPressure}
Pred :      OilPressure.time <= Now - 30*seconds
ER :        if Missed <= 2 then
              OilPressure.time = Now
              Missed = Missed + 1
              signal OilPressure_Warning
            else signal OilPressure_Alert
```

The enforcement rule specifies that if only one or two of the readings have been missed, a counter is incremented indicating that a reading has been missed and a warning is signaled using the exception `OilPressure_Warning`. If more than two readings have been missed, then an exception `OilPressure_Alert` is signaled, which might lead to a message being sent to the train operator. The counter `Missed` is reset to zero whenever a new sensor reading is written to attribute `OilPressure`.

### 3.1.4 Compatibility Function

The last component of an object, $CF$, is a compatibility function that expresses the semantics of simultaneous execution of each ordered pair of methods in the object. For each ordered pair of methods, $(m_i, m_j)$, a Boolean expression $(BE_{i,j})$is defined. $BE_{i,j}$ is evaluated to determine whether or not $m_i$ and $m_j$can execute concurrently. In many object-oriented systems, the execution of a single method of an object prevents any other methods of the object from being executed, i.e. the entire object is locked upon invocation of a single method. Through the use of the compatibility function, the designer of an object can allow more flexibility by defining the semantics of the compatibility of each pair of methods. By allowing a higher degree of concurrent access to the object through its methods, perhaps even relaxing serializability, the affected data may become imprecise. An in depth discussion of the semantic locking technique that utilizes the compatibility function to provide concurrency control to an object in RTSORAC can be found in [DW93].

Consider the following examples of compatibility function specifications:

```
CF(Get_OilPressure(), Get_OilTemp()) = TRUE
CF(Put_OilPressure(OP_reading), ShowLog(Log)) = (Log <> "OilPressure")
```

In the first example, the compatibility function is used to specify that the methods `Get_OilPressure` and `Get_OilTemp` of the **Train** object can always run concurrently (always `TRUE`). This is appropriate since these two methods operate on different attributes, `OilPressure` and `OilTemp`. The second example specifies that `Put_OilPressure` and `ShowLog` can run concurrently as long as the log to be displayed is not "OilPressure". If the requested log is "OilPressure", then the execution of the `ShowLog` method may be delayed or aborted.

## 3.2 Relationships

Relationships represent aggregations of two or more objects. In the RTSORAC model, a *relationship* (Figure 3.3) consists of $\langle N, A, M, C, CF, P, IC\rangle$. The first five components of a relationship are identical to the same components in the definition

$$\begin{array}{lcl}
\textbf{Relationship} & = & \langle N, A, M, C, CF, P, IC\rangle \\
\quad N & = & UniqueID \\
\quad A & = & \{a_1, a_2, ..., a_m\} \text{ where attribute } a_i = \langle N_a, V, T, I\rangle \\
\quad M & = & \{m_1, m_2, ..., m_n\} \text{ where method } m_i = \langle N_m, Arg, Exc, Op, OC\rangle \\
\quad C & = & \{c_1, c_2, ..., c_r\} \text{ where constraint } c_i = \langle N_c, AttrSet, Pred, ER\rangle \\
\quad CF & = & \text{compatibility function} \\
\quad P & = & \{p_1, p_2, ..., p_s\} \text{ where participant } p_i = \langle N_p, OT, Card\rangle \\
\quad IC & = & \{ic_1, ic_2, ..., ic_t\} \text{ where interobject constraint} \\
 & & \quad ic_i = \langle N_{ic}, PartSet, Pred, ER\rangle
\end{array}$$

Figure 3.3: Relationship Characteristics in RTSORAC

of an object. In addition, objects that can participate in the relationship are specified in the participant set $P$, and a set of interobject constraints is specified in $IC$.

Figure 3.4 illustrates an example of a **Energy Management** relationship for relating a **Train** object with a **Track** object. The **Track** object stores information such as track profile and grade, speed limits, maximum load, and power available. The energy management relationship uses both train and track information to determine fuel efficient throttle and brake settings.

### 3.2.1 Participants

$P$ is a set of participants in the relationship, each participant is of the form $\langle N_p, OT, Card\rangle$. $N_p$ is the name of the participant. $OT$ is the type of the object participating in the relationship. $Card$ is the cardinality of the participant, which is either *single* or *multi* [DG91]. Constraints can be used to express cardinality requirements of the relationship, such as minimum and maximum cardinality of the participants. In Figure 3.4, `Train` and `Track` are single cardinality participants.

### 3.2.2 Interobject Constraints

$IC$ is a set of interobject constraints placed on objects in the participant set, and is of the form $\langle N_{ic}, PartSet, Pred, ER\rangle$. $N_{ic}$, $Pred$, and $ER$ are as in object constraints, and $PartSet$ is a subset of the relationship's participant set $P$. The predicate is expressed using objects from the $PartSet$, allowing the constraint to be specified

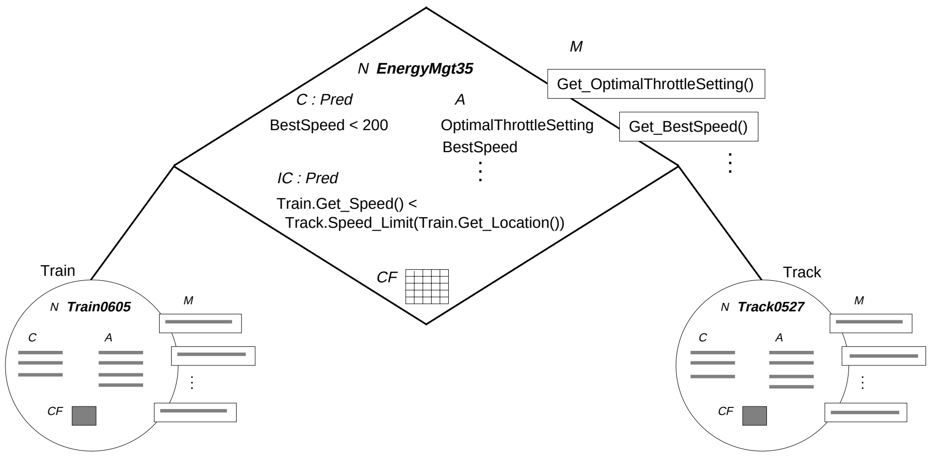


Figure 3.4: Example of **Energy Management** Relationship

over multiple objects participating in the relationship. Enforcement rules are defined as before by $\langle Exc, Op, OC \rangle$, however the operations $Op$ can now include invocations of methods of the objects participating in the relationship.

As an example of an interobject constraint, consider the **Energy Management**relationship in Figure 3.4. A **Train** object will be on one specific segment of track, represented by the **Track** object participating in the relationship. The train should obey the speed limits set on the track, so the following interobject constraint predicate could be specified:

$Pred:$ `Train.Get_Speed()` $<$ `Track.Speed_Limit(Train.Get_Location())`

If the speed of the train exceeds the speed limit posted at the train's location on the track, then the corresponding enforcement rule signals `SpeedLimitExceeded`.

## 3.3 Transactions

A *transaction* has six components, $\langle N_t, O, OC, PreCond, PostCond, Result \rangle$, where $N_t$ is a unique name or identifier, $O$ is a set of operations, $OC$ is a set of operation constraints, $PreCond$ is a precondition, $PostCond$ is a postcondition, and $Result$ is the result of the transaction. Each of these components is briefly described below.

| | | |
|---|---|---|
| **Transaction** | = | $\langle N, O, OC, PreCond, PostCond, Result \rangle$ |
| $N$ | = | $UniqueID$ |
| $O$ | = | $\{o_1, o_2, ..., o_m\}$ where $o_i$ is an operation |
| $OC$ | = | $\{oc_1, oc_2, ..., oc_n\}$ where $oc_i = \langle N_{oc}, OpSet, Pred, ER \rangle$ |
| $PreCond$ | = | Preconditions of the transaction |
| $PostCond$ | = | Postconditions of the transaction |
| $Result$ | = | Information returned by the transaction |

Figure 3.5: Transaction Characteristics in RTSORAC

### 3.3.1 Operations

$O$ is set of operations that represent the implementation of the transaction. These operations may include method invocations ($MI$), initiations of subtransactions, *commit* or *abort* statements, and statements for conditional branching, looping, and reads/writes on local variables. A subtransaction initiation allows for transactions to appear within the scope of other transactions. Method invocations ($MI$) are of the form $\langle MN, ArgInfo \rangle$, where $MN$ is the method name (prepended with the appropriate object id) and $ArgInfo$ is a set of tuples containing argument information. Each tuple is of the form $\langle aa, maximp, tcr \rangle$ where $aa$ is the actual argument to the method, $maximp$ is the maximum allowable imprecision of the argument, and $tcr$ is the temporal consistency requirement of the argument. The fields $maximp$ and $tcr$ are specified only for arguments that are used to return information to the transaction. These fields allow the transaction to specify requirements that differ from those defined on the data in the objects. For example, the transaction might be willing to accept a value whose temporal consistency requirements have been violated so as to meet other timing constraints. The data may still be useful to the transaction because of other available information (for example, it may be able to do some extrapolation). A transaction may also specify that data returned by a method invocation must be precise ($maximp$ is zero).

### 3.3.2 Operation Constraints

$OC$ is a set of constraints on the operations of the transaction. These constraints are of the same form as the operation constraints specified for methods, $\langle N_c, OpSet, Pred, ER\rangle$. As with methods, these constraints can be used to express precedence constraints, execution constraints, and timing constraints. For example, a transaction may require that a sensor reading which has been stored in the database be returned within two seconds.

### 3.3.3 Precondition, Postcondition, Result

$PreCond$ represents preconditions that must be satisfied before a transaction is made ready for execution. For example, it may be appropriate for a transaction to execute only if some specified event has occurred. The event may be the successful termination of another transaction, or a given clock time. $PostCond$ represents postconditions that must be satisfied upon completion of the operations of the transaction. The postconditions can be used to specify the semantics of what constitutes a *commit* of a transaction containing subtransactions. *Result* represents information that is returned by the transaction. This may include values read from objects as well as values computed by the transaction.

## 3.4 RTSORAC Features in RTSQL

The RTSORAC model provides a foundation for many of the language constructs found in RTSQL. This following paragraphs will highlight some of these features. Of particular interest is representation of data, constraints, and transactions.

**Data.** In the RTSORAC model, attributes in objects are used to represent the data items stored in the database. Recall that attributes have three fields for storing values associated with the data item. Currently, RTSQL provides support for storing two of these fields: the value field and the timestamp field. The value field will be stored as a column of a table. The mechanism used to store the value of the timestamp field is not specified in RTSQL, it is left to the underlying system to decide. RTSQL does provides a function for determining the timestamp value of an attribute. Future work in RTSQL will also provide support for the logical imprecision field.

Thus, the attributes of a RTSORAC object will be represented as columns in a RTSQL table. Each row of the table is used to store the values that would be associated with a single instance of a RTSORAC object.

**Constraints.** The RTSORAC model provides three types of constraints: constraints on data items, inter-object constraints, and operation constraints. Constraints on data items are modeled using a predicate and an enforcement rule. In RTSQL this corresponds to two constructs. The predicate of the RTSORAC constraint is roughly equivalent to a constraint specification in RTSQL. Constraints in RTSQL state the logical and temporal consistency requirements of the data using a boolean expression. The RTSORAC enforcement rule is roughly equivalent to the condition handling mechanism of RTSQL. It provides a means of specifying some action to be taken if a constraint violation should occur.

The RTSORAC inter-object constraints are used to specify constraints on groups of objects. They also consist of a predicate and an enforcement rule. In RTSQL, the predicate of an inter-object constraint would be equivalent to specifying a constraint over groups of tables. Since RTSQL extends SQL, and SQL provides allows constraints to extend over more than one table, RTSQL can support inter-object constraints of RTSORAC.

Operation constraints in RTSORAC are used to express precedence constraints, execution constraints, and timing constraints. The most developed support in RTSQL is for timing constraints. Some support is provided for execution constraints and precedence constraints. RTSORAC operation constraints consist of a predicate and an enforcement rule. The predicate for timing constraints in RTSQL will appear as a timing constraint clause on an SQL statement or block. For example, to express a deadline for completion of a particular statement, RTSQL provides a "`COMPLETE BEFORE time-exp`" clause. As in constraints on data items and inter-object constraints, the enforcement rule is implemented in RTSQL using the condition handling mechanism.

**Transactions.** RTSQL supports most of the features of RTSORAC transactions. RTSQL provides a mechanism for specifying a series of operations for the transaction. The operation constraints are the same as those discussed in the previous paragraph. RTSQL provides constructs for specifying pre-conditions and post-conditions for transactions. The operations of the transaction will produce the results as specified in the RTSORAC transaction model. RTSQL does not support imprecision in the parameters of a transaction, nor imprecision in any of the results.

# Chapter 4

# Language Extensions

In this chapter we present the syntax and semantics of initial extensions to SQL2 to support real-time database systems. We call the resulting extended language RTSQL. These extensions to SQL2 appear in three areas. The first area is how the notion of SQL2 *constraints* is extended for both data and execution. Constraints are used to specify the semantics of correctness (including the aspect of time) of data, operations, and transactions. The second area is the addition of a new construct called a *directive.* A directive is used to specify assertions about data, operations, and transactions. The third area is *transaction structure.* Transaction structure addresses what information will be presented to the system on behalf of a transaction. Each of these areas is discussed in the following sections.

## 4.1 Constraints

Since real-time databases essentially add the notion of timing constraints to conventional databases, constraints are a primary focus of the extensions provided in RTSQL. Recall that constraints in SQL2 are mechanisms for specifying the logical consistency requirements of data. In RTSQL, the notion of constraints is extended in two ways. First, temporal consistency requirements of the data can be specified. Second, data manipulation statements and transactions can have timing constraints specified upon their execution.

### 4.1.1 Temporal Consistency of Data

Typical database systems provide for the specification of constraints and recovery from their violations. In this section we discuss how RTSQL provides for the specification and recovery associated with data temporal consistency constraints. In addition, this section discusses how data temporal consistency violations are detected.

**Specification of Data Temporal Consistency Constraints.** In RTSQL, data constraint definitions are extended to allow for specification of temporal consistency requirements of the data. These requirements are usually expressed by indicating the maximum acceptable age for a data item. Computation of the age of a data item requires that the system record the time that the data value was determined (perhaps it is the time the value was generated by a sensor, or the time that the value was written). Since it would not be necessary to determine the age of every data item in the database, the following RTSQL clause can be specified during definition of a data item to specify that a timestamp will be required:

```
<data timestamp clause> ::=
  [WITH TIMESTAMP <datetime type>]
```

Note that `<datetime type>` is a type provided by SQL2. Access to this timestamp value is through a function on the data item called `TIMESTAMP`. For example, the timestamp value on the data item `temp_reading` would be accessed as `TIMESTAMP(temp_reading)`. Also note that SQL2 provides a function that returns the current time called `CURRENT_TIMESTAMP`. SQL2 provides syntax for constraint specification that can be used to specify temporal consistency requirements when used in conjunction with `CURRENT_TIMESTAMP` and the `TIMESTAMP` function. For example, the following constraint, `temp_reading_avi`, specifies that for the data item `temp_reading` to be absolutely temporally consistent, it must be less than ten seconds old:

```
CONSTRAINT temp_reading_avi
  CHECK (CURRENT_TIMESTAMP - TIMESTAMP(temp_reading)) DAY to SECOND
         < INTERVAL '10' SECOND
```

Here, the SQL2 CHECK clause contains a boolean expression which computes the age of temp_reading and determines whether it is less than 10 seconds old. SQL2 does not allow constraint specifications to include references to any of the functions that return dates and times (such as CURRENT_TIMESTAMP) [DD92]. From the example shown, it is obvious that this restriction must be relaxed in RTSQL.

Relative temporal consistency among data items can be expressed by comparing their timestamps. For example, the following constraint speed_bearing_rvi specifies the relative temporal consistency requirements of speed and bearing:

```
CONSTRAINT speed_bearing_rvi
  CHECK TIMESTAMP(speed) BETWEEN
        TIMESTAMP(bearing) - INTERVAL '2' SECOND
          AND TIMESTAMP(bearing) + INTERVAL '2' SECOND
```

Here, the speed timestamp is checked to see if it is within two seconds of the bearing timestamp.

Constraints themselves may be valid only for a given period of time. The following RTSQL clauses can be specified as part of a constraint specification to indicate when a constraint is active:

```
<constraint validity interval clause> ::=
   [AFTER <datetime value expression>]
   [BEFORE <datetime value expression>]
```

For example, the previous speed_bearing_rvi constraint is specified to be active only after CONTACT_MADE, where CONTACT_MADE is of a *datetime* data type:

```
CONSTRAINT speed_bearing_rvi
  CHECK TIMESTAMP(speed) BETWEEN
        TIMESTAMP(bearing) - INTERVAL '2' SECOND
          AND TIMESTAMP(bearing) + INTERVAL '2' SECOND
  AFTER CONTACT_MADE
```

Note that the <constraint validity interval clause> may be applied to any constraint specification.

**Detecting Data Temporal Consistency Violations.** A violated data temporal consistency constraint is an indication that the data did not get refreshed as expected. In such situations, the database system should react. A simple response might be to refresh the data by causing the appropriate update action to be executed.

The main issue with respect to data temporal consistency is how to determine when the constraints are evaluated. One technique represents the *passive* approach, where the database system waits until the data value is read, and checks the constraint at that time. The second technique is the *active* approach, where the database system checks the constraint periodically. The active approach could be implemented through the use of timers, where timers are set (or reset) when data values are written. When a timer expires, it indicates that the next update did not occur. The active approach carries more overhead, especially if there are a large number of data items, and their update intervals are relatively short. Constraints in RTSQL that specify the temporal consistency requirements of data use the passive approach for detecting violations. The active approach could be supported through an extended version of the trigger mechanism specified in SQL3.

**Recovery from Data Temporal Consistency Violations.** When the constraint violation is detected, a repair action must be taken. SQL2/PSM provides basic condition handling facilities (see Section 2.2.1), which must be extended to handle data temporal consistency violations.

The current version of the SQL2/PSM condition handling mechanism does not provide a means of associating a named constraint with an exception. When a named constraint is violated in SQL2/PSM, a generic exception condition is raised. At this point, a `GET DIAGNOSTICS` statement could be used to determine which constraint was actually violated, and allow the user to specify the appropriate corrective action. A more direct approach is to allow the user to associate an exception with the violation of a named constraint. When the constraint is violated, the exception is raised. This approach allows the user to write exception handlers specific to each constraint specified.

Since constraints in RTSQL are supported using the passive approach, constraint violations will be detected when a data value is read. This means that the exception will be raised by a statement that may have a corresponding exception handler available. Such a constraint violation could occur if the sensor supplying the data malfunctions, or the transaction responsible for the update misses its deadline. An exception handler could attempt to update the data value or it may simply signal the user that a sensor check should be performed.

### 4.1.2 Timing Constraints on Execution

Typical database systems do not provide any mechanisms for placing timing constraints on statements or transactions. Timing constraints on execution are used to define the semantics of what constitutes the correct execution of a statement with respect to time. In this section we discuss how RTSQL extends constraints to provide this functionality.

**Specification of Execution Timing Constraints.** RTSQL specifies time constrained execution by placing timing constraints on individual data manipulation statements or, as appears in SQL2/PSM, a block of statements. This specification uses the following clauses:

```
<timing constraint clause> ::=
  [START BEFORE <datetime value expression>]
  [START AFTER  <datetime value expression>]
  [COMPLETE BEFORE <datetime value expression>]
  [COMPLETE AFTER  <datetime value expression>]
  [PERIOD <interval value expression>
    [START AT <time expression>]
    [UNTIL <boolean expression>]]

<datetime value function> ::=
  CURRENT_DATE | CURRENT_TIME[<time precision>] |
  CURRENT_TIMESTAMP[<timestamp precision>]
```

The START BEFORE and COMPLETE BEFORE clauses are used to express the latest start time and latest finish time for the execution of the statement. The START AFTER and COMPLETE AFTER clauses are used to express the earliest start time and earliest finish time for the execution of the statement. The PERIOD clause allows for the establishment of a periodic execution of a statement. The START AT portion of the PERIOD clause establishes the period frame. The UNTIL portion of the clause allows for the specification of the conditions that must be met before periodic execution may terminate.

Recall that in SQL2, *datetime* valued expressions have been defined, and can include references to *datetime* value functions. In RTSQL, if such functions are included in an expression, they are all evaluated before the statement begins execution. Further, all of the occurrences of the *datetime* value functions in a statement will appear to have been evaluated at the same instance of time. This holds true for nested statements, where a compound statement may contain statements or other compound statements.

For example, suppose we have the following:

```
X:BEGIN
   SELECT price FROM stocks WHERE name="Acme"
     COMPLETE BEFORE CURRENT_TIMESTAMP + INTERVAL '30' SECOND;
   -- other computations
END X COMPLETE BEFORE CURRENT_TIMESTAMP + INTERVAL '1' MINUTE;
```

The execution timing constraint on the SELECT statement specifies that it must complete execution within 30 seconds. The timing constraint on the compound statement specifies that it must complete execution within 1 minute. Note that the value of CURRENT_TIMESTAMP will be the same for both timing constraints (since they appear in the same compound statement).

**Detecting Execution Timing Constraint Violations.** Detecting an execution timing constraint violation can be done through the use of timers. When a statement is encountered, the timing constraints are evaluated, and timers are set for the various types of timing constraints. For example, suppose a statement contains the following clause:

```
START BEFORE CURRENT_TIMESTAMP + INTERVAL '1' MINUTE
```

This timing constraint indicates that the statement should begin execution within 1 minute. Note that statement might not begin immediate execution because some of the resources it requires are unavailable.

**Recovery from Execution Timing Constraint Violations.** When an execution timing constraint is violated, an exception condition is raised. For each constraint alternative that can be specified in the `<timing constraint clause>` of Section 4.1.2, there will be a corresponding system-defined `SQLSTATE` value. Thus, a user can provide an exception handler that specifies the compensating actions that should be performed when the timing constraint violation occurs. Note that the execution of the exception handler itself falls within the timing constraint of the block, so the block should be designed to allow for that possibility.

The user must utilize the timing constraints and condition handlers in a fashion that supports the timing needs of an application. For example, a financial application for stock trading may require that a decision for initiating a stock trade must occur within 5 minutes. This action may involve extracting information from the database and performing some computation. In designing the code for this, the user must be conscious of where to place the timing constraints to meet the needs of the application. They may wish to place timing constraints on the statements which acquire the stock information, so that if they cannot be completed by the given deadline, the exception handler may still have time to execute some compensating action. For example:

```
Y:BEGIN
  DECLARE EXIT HANDLER FOR SQLSTATE VALUE 'TC059'
    BEGIN
      -- compensating action if compound stmt X not completed in time
    END

  X:BEGIN
    DECLARE EXIT HANDLER FOR SQLSTATE VALUE 'TC059'
      BEGIN
        -- compensating action if Acme stock price not found in time
      END
      SELECT price FROM stocks WHERE name="Acme"
        COMPLETE BEFORE CURRENT_TIMESTAMP + INTERVAL '1' MINUTE;

      -- other computations

    END X COMPLETE BEFORE CURRENT_TIMESTAMP + INTERVAL '5' MINUTE;

  END Y;
```

The code 'TC059' is used to represent a predefined condition value for the violation of a `COMPLETE BEFORE` timing constraint. Condition handlers have been declared to provide a compensating actions if the constraints are violated. Note that the handler declared in the compound statement X is intended to provide a compensating action if the timing constraint on the `SELECT` statement is violated. The handler declared in the compound statement Y is intended to provide the compensating action if the timing constraint is violated for the compound statement X.

### 4.1.3 Condition Handlers

The previous example demonstrates a weakness in the condition handling mechanism of SQL/PSM. Note that the user had to specify the `SQLSTATE` value to indicate which timing constraint was being handled. This can be awkward for the user in both writing the handlers, and maintaining them later since it is unlikely they will remember these codes. RTSQL provides a set of predefined constraint names to alleviate this problem. Also, to support reusability of code, a routine may be specified in place of the compound statement. RTSQL provides the following syntax for condition handlers (note that this is a modification of the syntax proposed in SQL/PSM):

```
<handler declaration> ::=
  DECLARE <handler type> HANDLER
    FOR <condition value list>
    <handler action>

<handler type> ::=
  CONTINUE | EXIT | REDO | UNDO

<handler action> ::=
    <routine name> [<argument list>]
```

```
<condition value list> ::=
  <condition value> [{<comma><condition value>}...]

<condition value> ::=
    SQLSTATE [VALUE] <sqlstate list>
  | <constraint name>
  | <system defined condition name>

<sqlstate list> ::=
  <character string literal> [{<comma><character string literal>}...]

<system defined condition name> ::=
    SB_CONSTRAINT
  | CB_CONSTRAINT
  | TC_CONSTRAINT
```

Three RTSQL system-defined constraints have been defined, `SB_CONSTRAINT` , `CB_CONSTRAINT`, and `TC_CONSTRAINT`. The constraint `SB_CONSTRAINT` is used for violation of the `START BEFORE` clause. `CB_CONSTRAINT` is used for violation of the `COMPLETE BEFORE` clause. `TC_CONSTRAINT` is used when the system detects a timing constraint conflict. For example, if `START AFTER` is earlier than `START BEFORE` on the same action, then this constraint will be violated. Further discussion of the constraint is in Section 5.2.5.

Also note `<constraint name>` as another possibility for a condition value. This corresponds to the user-defined constraints. For example,

```
  DECLARE EXIT HANDLER FOR stock.check_id my_handler();
```

declares an `EXIT` handler for the violation of the `check_id` constraint defined in the table `stock`. A handler routine `my_handler` is specified as the action to be performed if the `check_id` constraint is violated.

## 4.2 Directives

In order to support the predictability requirement of real-time databases, RTSQL introduces a concept called a *directive*. Directives provide information to the database system to facilitate maintenance of the constraints and predictability. For example, a transaction may contain a constraint that specifies that a temperature data item must be retrieved within milliseconds for the transaction to produce correct results. Since data fetched from disk has unpredictable access time, a directive could specify that the temperature data item should be maintained in main memory at all times, thus making its data retrieval time faster and more predictable. Note that the directive may not be sufficient to insure that the constraint will always be met. For example, if the temperature data is unavailable due to locking, or is temporally inconsistent, it may not be accessible within the time period specified by the transaction. That is, directives are not sufficient for enforcing logical or temporal correctness.

Directives differ from constraints in that constraints address the logical and temporal consistency requirements for data and operations. The applications that utilize the database system determine these consistency requirements, which in turn are mapped to constraints. For example, a speed data item may have logical consistency requirements that specify that it must always be positive, and temporal consistency requirements which specify that it must be less than 5 seconds old. Directives provide additional information to the database system to facilitate maintenance of the constraints and predictable access time to data. As such, directives may involve hardware characteristics. For example, a sensor value may be located at a particular address in memory. It may be more efficient to simply notify the database system of the location of the data value so that it can use the value directly. The following paragraphs feature the directives proposed for RTSQL.

**Data Storage.** One characteristic that can have an affect on transaction processing and timely data availability is how and where the data is stored by the system. For example, if a data value must be accessible within milliseconds, it might be necessary for the database system to maintain that data in main memory. Similarly, if a table of data items must have a bounded search time, then a maximum number of entries in the table could be specified. RTSQL storage directives are used to allow programmers the ability to specify where and how a data item or table is to be stored. The following clause, part of a table definition, is used to specify storage requirements:

```
<storage clause> ::=
  [STORE IN <storage type> [AT <location>]]
```

where the domains of `<storage type>` and `<location>` are architecture-dependent. For example: `STORE IN main_memory` could be used in a SQL table definition to specify that the table be stored only in main memory. The `AT <location>` clause could be used to store the table at a particular location in main memory.

To allow determination of an upper bound on the time it takes to access a table, the following RTSQL directive clause can be specified during the definition of a table:

```
<table size clause> ::=
   [SIZE UPPER LIMIT <integer>]
```

indicating that this is the maximum number of data items that can be in this table.

**Relative Importance Level.** This directive allows for the specification of the relative importance of an action. A scheduling algorithm may use relative importance of tasks as a parameter in determining scheduling priority of the tasks. Not all systems will utilize this directive, and as such, would be free to ignore this directive upon notification. Also, the semantics of the various levels may vary in different systems, hence the portability of this directive is limited. The importance directive clause is as follows:

```
<importance clause> ::=
  IMPORTANCE LEVEL <importance level>
```

**Asynchronous Execution.** In real-time applications, it may be useful to notify the system that some actions can be done asynchronously. In systems with multiple processors available, this may make it easier for the system to meet given timing constraints. Applications which utilize real-time databases take advantage of this capability by identifying which actions can be done in parallel. This concept of asynchronous statement execution has been proposed in SQL3[MS92]. Specification of asynchronous execution of a statement in SQL3 is done using the following clause:

```
[ ASYNC ( <async statement identifier> ) ]
```

If left unspecified, the default is synchronous. SQL3 also provides a statement for testing the completion of an asynchronous statement:

```
<test completion statement> ::=
    { TEST | WAIT }
    { ALL | ANY | <async statement identifier list> } COMPLETION
```

The `{ }` notation indicates alternatives. For example, the `<test completion statement>` must begin with either the `TEST` or `WAIT` keyword. The `TEST` alternative is used to check to see if asynchronous statements have completed execution. If they have not, an exception condition is raised. The `WAIT` alternative is used to wait for the asynchronous statements to complete execution. If the asynchronous statements have already completed execution when the `WAIT` statement is executed, an exception condition will be raised. The second clause specifies which asynchronous statement the TEST or WAIT is intended for. There are three alternatives: `ALL` is for all asynchronous statements in the transaction, `ANY` is for any one of the asynchronous statements in the transaction, and the third option is to list the particular asynchronous statements in the transaction. Note that RTSQL requires no changes from the asynchronous execution specification proposed for SQL3; RTSQL simply endorses the extension as necessary for real-time databases.

**Worst Case Execution Time.** RTSQL also allows for directives that specify the worst case execution time (`WCET`) of a statement. This value is made available to the system by the user, who has determined this value through analysis[PM94]. This value can then be utilized by the system in a variety of ways. For example, `WCET` can be used to determine feasible task schedules [LL73] or to determine if an action will be able to complete before its deadline. In these instances, the database system may be able to predict that a timing constraint will be violated before it actually happens.

```
[ WCET ( <time literal> ) ]
```

## 4.3 Transaction Structural Specification

As described in Section 2.2.2, conventional ACID transactions are often not suitable for real-time database systems since their rigidity overly restricts real-time scheduling options. The RTSQL transaction structure is designed to relax ACID requirements by allowing transactions to be composed of subtransactions. Subtransactions may individually preserve one or more of the ACID properties while the parent transaction does not necessarily preserve all of the ACID properties. These capabilities allow a transaction designer to selectively determine which ACID properties to enforce on parts of the transaction by structuring it using appropriate subtransactions [FWP94].

### 4.3.1 RTSQL Flexible Transaction Concepts

To support this ability to design flexible transactions, the RTSQL specification of a transaction includes: a *precondition part*, a *specification part*, a *body*, a *postcondition part*, and a *recovery* part (Figure 4.1). The precondition consists of requirements that define the conditions that the transaction must satisfy to begin execution. The specification part is used to define data structures, timing requirements, resource limitations, data dependencies, transaction criticality, atomicity of the transaction, preemptability of the transaction, and execution dependencies. The body of the

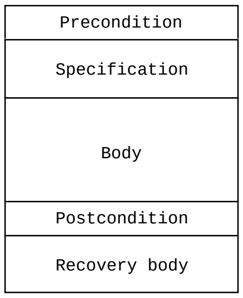


Figure 4.1: Abstract View of Flexible Transaction Structure

transaction includes database access code and transaction processing code. The postcondition contains predicates that define what constitutes a correct execution of the transaction. The recovery part contains semantically defined recovery mechanisms for defined errors. If this part is omitted, a default recovery can be chosen at database definition time.

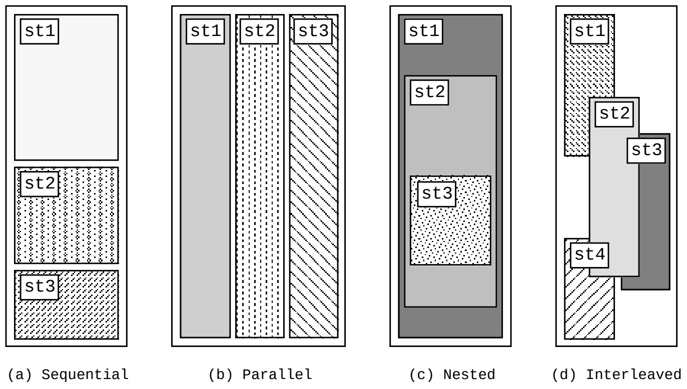


Figure 4.2: Flexible Subtransaction Structure

There are four basic structural building blocks for decomposing transactions into subtransactions: sequential, parallel, nested, and interleaved. For example, if we want the simple serial string of subtransactions (Figure 4.2a) we model this by specifying that the subtransaction boundaries do not overlap (the precondition for one subtransaction is that the predecessor subtransaction has completed) and that commit is allowed upon completion of a subtransaction. This specification causes the transaction to behave as a set of disjoint subtransactions that commit serially. Such a structure could also be used to model a long duration transaction [GMGK+91].

The transaction structure also allows the specification of concurrent or parallel executions (Figure 4.2b). Subtransactions can have the precondition that all must begin at the same time. Conversely, the structure allows specification of a set of disjoint parallel subtransactions being commit-related, by using postconditions that disallow commit until all are ready. Using a similar specification, the structure can also specify multi-level nested transactions by appropriately placing subtransaction boundaries and by using pre and postconditions on executions (Figure 4.2c). To specify a more complex interleaved model, the structure allows placement of subtransaction *begin*s where needed. This specification includes the partition of the database on which to operate and under what conditions the subtransaction can commit relative to interleaved siblings (Figure 4.2d).

The recovery part of the transaction specification allows semantically defined recovery. Conventional transaction recovery is based on recovering to a past state. In real-time the past is gone, which implies a need for a forward recovery. Since the application designers know how data is being used, recovery should be defined by the application's needs. For example, in a periodic real-time system, recovery back to an old state may be inappropriate. Instead, the transaction may specify recovery to be performed by delaying restart until the next update cycle. Recovery may require blocking other transactions until a new update, or forward recovery to move to a present consistent state.

The flexible transaction structure allows the transaction applications writers to select the degree of consistency and correctness they need from the database, which can improve data availability. Simulation results verify that improvements can be realized if transactions are decomposed into subtransactions whose commit dependencies are specified in pre and postconditions [For93]. Although the RT-SQL specifications can be ad hoc, there are methodologies that can be applied to guarantee that the database's consistency is not violated [For93, Sha85].

### 4.3.2 RTSQL Flexible Transaction Specification

To specify the above transaction structuring, mechanisms and constraint definitions that allow for control over transaction execution and commit must be defined. Since SQL2 does not provide any mechanism for writing named transactions, RTSQL provides the following syntax:

```
<transaction-specification> ::=
  TRANSACTION <transaction name> (<input parameters>)
    [<local declarations>]
    [[REPLACEABLE] PRECONDITION <boolean expression>]
    [[REPLACEABLE] POSTCONDITION <boolean expression>]
    [[REPLACEABLE] RECOVER ON <condition>
      [AUTO | SEMI <transaction name> | MANUAL]]
    [[REPLACEABLE] ISOLATION LEVEL <level of isolation>]
    [[REPLACEABLE] ACCESS MODE <transaction access mode>]
    [[REPLACEABLE] DIAGNOSTICS SIZE <number of conditions>]
  BEGIN
    <transaction body>
    //A COMMIT or ROLLBACK statement must terminate execution of the transaction
  END
```

The PRECONDITION clause allows specification of a predicate that must evaluate to true before the transaction may begin execution. The POSTCONDITION clause allows specification of a predicate that is evaluated before the COMMIT statement, and can be used to define what constitutes the correct execution of the transaction. If the predicate evaluates to true, then the COMMIT statement is executed. If the predicate evaluates to not true[1], the transaction is aborted. The RECOVER clause specifies the conditions under which the recovery should be performed, and whether the recovery should be automatic, semi-automatic (with a recovery transaction specified), or manual. The input parameters to the transaction allow information to be passed to the transaction from the invoking entity.

The clauses ISOLATION LEVEL, ACCESS MODE and DIAGNOSTICS SIZE, are basically the same as those defined in the SQL2 SET TRANSACTION statement. ISOLATION LEVEL allows specification of the transaction's isolation level, as described in the ACID properties. Four levels of isolation are available in SQL2: SERIALIZABLE, READ COMMITTED, READ UNCOMMITTED, and REPEATABLE READ. ACCESS MODE is specified as either READ ONLY or READ WRITE. A READ ONLY transaction that attempts to do a write will abort and generate an error. The clause DIAGNOSTICS SIZE specifies the size of the area used for storing errors reported by the system. Efforts within SQL3 are examining mapping of error codes in the diagnostics area to exceptions [Gal92].

Note that many of the clauses may be preceded by the keyword REPLACEABLE. This is to allow the transaction writer to specify whether or not a particular characteristic of the transaction can be replaced at a later time through the use of the SET TRANSACTION statement. SQL2 provides such a statement for setting the isolation level, access-mode, and the size of the diagnostics area, with syntax similar

[1]We use not "not true" due to SQL2's three valued logic.

to that shown above. To be consistent with the transaction specification, the `SET TRANSACTION` statement in SQL2 is modified in RTSQL as follows:

```
<set transaction> ::=
  SET TRANSACTION <transaction name>
    [PRECONDITION  [AND | OR] <boolean expression>]
    [POSTCONDITION [AND | OR] <boolean expression>]
    [RECOVER ON <condition>
      [AUTO | SEMI <transaction name> | MANUAL]]
    [ISOLATION_LEVEL <isolation-level>]
    [ACCESS_MODE <access-mode>]
    [DIAGNOSTICS SIZE <value spec>]
```

Any feature of the transaction specification that is replaceable can be changed using the `SET TRANSACTION` statement. If the precondition or postcondition is replaceable, then a new boolean expression can be provided. If the `OR` keyword is specified before the new boolean expression, the intent is to perform a disjunction of the provided boolean expression with the original boolean expression. The `AND` keyword works in a similar fashion. Note that the `OR` keyword allows an alternative condition, while the `AND` clause allows for an additional condition.

Since SQL2 has no transaction initiating statement, RTSQL provides the following syntax:

```
START TRANSACTION <transaction name> (<input arguments>)
```

When the `START TRANSACTION` statement is executed, the information contained in the input arguments is passed to the named transaction to start its execution. Part of this process involves checking the precondition of the transaction, and if the precondition evaluates to true, returning a transaction identifier (*tid*) that can be used to refer to that transaction execution. If the precondition does not evaluate to true, or the system is unable to execute the transaction, the transaction identifier returned will be `NULL`. Future work will address the use of exceptions and other mechanisms to help the transaction writer react to such occurrences. Also note that since `START TRANSACTION` is just another statement, all of the timing and execution constraints mentioned in Section 4.1.2 apply here. Thus entire transactions can also be time constrained.

```
TRANSACTION main ()
  tid st1, st2, st3;
  POSTCONDITION
    (((ECOMMIT(st1) <> UNDEFINED) AND (ECOMMIT(st2) <> UNDEFINED)) OR
     ((ECOMMIT(st1) <> UNDEFINED) AND (ECOMMIT(st3) <> UNDEFINED)) OR
     ((ECOMMIT(st2) <> UNDEFINED) AND (ECOMMIT(st3) <> UNDEFINED)))
BEGIN
  ...
  st1 := START TRANSACTION trans1();
  st2 := START TRANSACTION trans2();
  ...
  st3 := START TRANSACTION trans3();
  COMMIT;
END;
```

Example 1: Sequential Transactions

In SQL2, *datetime* valued expressions have been defined, and can include references to *datetime* value functions. To better support transaction structuring, RT-SQL extends the available *datetime* value functions to include `EINITIATE`, `ESTART`, `ECOMPLETE`, `ECOMMIT`, and `EABORT`. These functions take a transaction identifier as an argument and return a datetime value. `EINITIATE` returns the time the transaction was initiated by a user or another transaction. `ESTART` returns the time the transaction started execution in the system. `ECOMMIT` returns the time that the transaction committed. `EABORT` returns the time that the transaction aborted. `ECOMPLETE` returns the time that the transaction either committed or aborted. If the event has not yet occurred, the functions will return an infinite time value called `UNDEFINED`. *Datetime* valued expressions and interval expressions are extended in RTSQL to accommodate this value appropriately.

### 4.3.3 Examples of RTSQL Flexible Transaction Structures

With these extensions, RTSQL now has enough expressive power to model sequential transactions, nested transactions, and interleaved transactions. Example 1 demonstrates sequential execution with commit dependencies. In this example, the transaction `main` will only commit if two of the three subtransactions commit.

Example 2 shows nested transactions. In this example, transaction `main` is a parent transaction, and `trans1` is a child transaction. With respect to `main`, `trans2` is a grandchild, and so forth. Note that a parent transaction can only see its children, and not its children's descendents. Hence the parent's commit dependencies in the postcondition can only be expressed on children transactions, not their children's descendants.

```
TRANSACTION trans1()
  tid st;
BEGIN
  st := START TRANSACTION trans2();
  ...
  COMMIT;
END;

TRANSACTION main ()
  tid st;
  REPLACEABLE POSTCONDITION
   (ECOMMIT(st) <> UNDEFINED))
BEGIN
  st := START TRANSACTION trans1();
  ...
  COMMIT;
END;
```

```
TRANSACTION trans2()
  tid st;
BEGIN
  ...
  st := START TRANSACTION trans3();
  COMMIT;
END;

TRANSACTION trans3()
BEGIN
  ...
  COMMIT;
END;
```

Example 2: Nested Transactions

Example 3 shows interleaved transactions. With interleaved transactions, events, such as the commit of subtransactions, are considered important. In this example, subtransaction `trans2` can only start after subtransaction `trans1` has started execution, and before `trans1` completes execution. Subtransaction `trans3` can only start after the commit of `trans1` (*st1*). Note that there is also a `START BEFORE` clause for `trans3` that specifies that the transaction should start within 2 minutes of when the `START TRANSACTION` statement started execution. If `trans1` (*st1*) never commits, the `START BEFORE` clause will be violated, and `trans3` will never start execution. Also note that since `trans3` is started synchronously, it will block while waiting for `trans1` to complete execution.

Example 4 shows three subtransactions that may execute in parallel. Note that

```
TRANSACTION main ()
  tid st1, st2, st3;
  REPLACEABLE POSTCONDITION ((ECOMMIT(st1) <> UNDEFINED) OR
                             (ECOMMIT(st3) <> UNDEFINED))
BEGIN
  st1 := ASYNC START TRANSACTION trans1();
  st2 := ASYNC START TRANSACTION trans2()
           START AFTER ESTART(st1)
           START BEFORE ECOMPLETE(st1);
  st3 := START TRANSACTION trans3()
           START AFTER ECOMMIT(st1)
           START BEFORE EINITIATE(st3) + INTERVAL '2' MINUTE
           FINISH BEFORE EINITIATE(st3) + INTERVAL '5' MINUTE;
  COMMIT;
END
```

Example 3: Interleaved Transactions

```
TRANSACTION main ()
  tid st1, st2, st3;
BEGIN
  BEGIN
    st1 := ASYNC START TRANSACTION trans1();
    st2 := ASYNC START TRANSACTION trans2();
    st3 := ASYNC START TRANSACTION trans3();
  END;
  COMMIT;
END;
```

Example 4: Asynchronous Subtransactions

asynchronous execution is not sufficient to specify that the transactions will start and end at the same time, it simply specifies that they may run in parallel.

## 4.4 Discussion

This chapter presented the syntax and semantics of initial extensions to SQL2 to support real-time database systems. The extensions appeared in three areas: constraints, directives, and transactions. In developing these extensions, a considerable effort was made to use existing constructs within the language. This was most evident in the area of data temporal consistency constraints and transactions. Support for data temporal consistency constraints required only a few modifications to SQL including timestamped values (and a function to reference the value) and allowing datetime functions to appear in constraint expressions. SQL2 transactions already had a `SET TRANSACTION` statement available for setting the characteristics of a transaction. This statement was extended to address the additional features of RTSQL transactions including preconditions, postconditions, and a recovery level.

Other far more extensive changes to SQL2 were also required by RTSQL. Directives, which address the resource utilization limits required in real-time database systems, is a totally new concept to SQL2. We expect directives to be a rather controversial addition to a language which traditionally has left it to the implementation to make many of the decisions of how an action is performed and how its resources are managed. Transactions are another area within SQL2 which are considered controversial. RTSQL has constructs for named parameterized transactions, which then allows for subtransactions (nested transactions) to be specified. In SQL2, transactions are not named, and the user has no control over when a transaction starts execution. SQL3 has started to address this issue, and does support a simple `START TRANSACTION` statement. The standards committee views transactions as an area of SQL that needs extensive work.

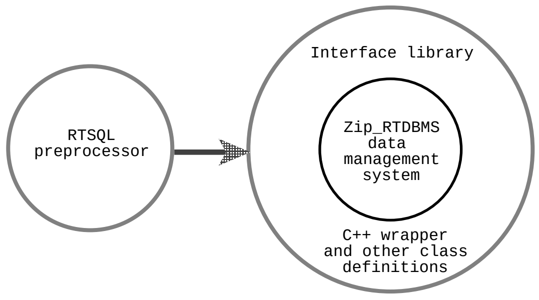
The addition of timing constraints to actions is also a new concept to SQL2. The constructs in RTSQL to support timing constraints are done as simple clauses which can be added to existing statements. At this point in time, these clauses do not change the semantics of these statements except with respect to placing time constraints on their execution. To support condition handlers for violations of these constraints required the addition of SQLSTATE values and system defined condition names. All of these additions have been viewed as useful to SQL2 by the standards committee, even in the context of systems which are not real-time.

One area which is not addressed by RTSQL, yet appears in the DISWG requirements, is concurrency control criteria and imprecision. Work in the area of bounded imprecision in transaction processing [RP95, DP93, DiP95] is relatively new area of research. Though it would be relatively easy to add imprecision fields to data values in RTSQL, and to allow the expression of constraints that utilize these values, considerable effort is still required in studying the implications of these additions to the underlying database system. For example, the concurrency control criteria that is used by the system is tightly bound with how and when the imprecision is accumulated in the data values. Concurrency control criteria and imprecision are also areas which are considered relatively new and controversial by the standards committee.

# Chapter 5

# Implementation

This chapter describes the prototype system that was constructed to show the feasibility of the timing constraint specifications of RTSQL. The system consists of a RTSQL preprocessor, the real-time data manager Zip_RTDBMS, and an interface library we developed to facilitate the use of Zip_RTDBMS. The RTSQL preprocessor translates C++ programs containing RTSQL statements into C++ programs that use the interface library. The architecture of the prototype system is shown in Figure 5.1. The implementation of each major portion of the system will be discussed in the following sections.



RTSQL preprocessor
Interface library
Zip_RTDBMS data management system
C++ wrapper and other class definitions

Figure 5.1: System Architecture

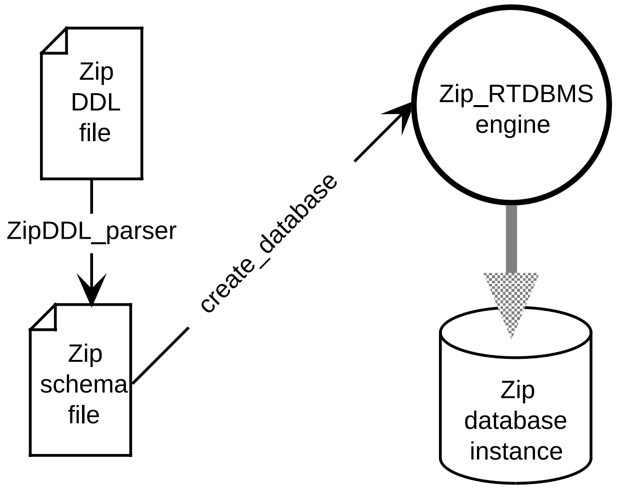


Figure 5.2: Zip_RTDBMS Data Definition

## 5.1 Interface to Zip_RTDBMS

The interface to Zip_RTDBMS consists of C++ class definitions that simplify access to the Zip_RTDBMS data manager. The next section will provide a brief description of Zip_RTDBMS. Section 5.1.2 will discuss the implementation of the interface library.

### 5.1.1 Zip_RTDBMS

Zip_RTDBMS is a memory resident, high speed, real-time data management system developed by DBx, Inc. It provides facilities for storing and retrieving data with deterministic response times. Zip_RTDBMS supports the static definition of a schema, which includes resource requirements and access behavior. It also provides a low level call interface that supports connection management, data management, and general error management.

In Zip_RTDBMS, the schema specifies the relations contained in a database, along with their type and maximum size. Three types of relations are supported: STATIC, BOUNDED, and ROLLING. STATIC relations are for data that changes minimally through the lifetime of the database. ROLLING relations are for time-varying data, and can be viewed as circular buffers with fixed capacity. ROLLING tables are intended to handle data streams such as those associated with sensors, where the data stored in the table will represent an interval of time, the oldest data being replaced by the newest set of values. BOUNDED relations are basically the same as relations found in traditional database systems except that they have a maximum fixed size.

Figure 5.2 shows the steps necessary to create a Zip_RTDBMS schema and database instance. The user creates a file containing Zip_RTDBMS data definition commands (Zip DDL file). This file is parsed and compiled by the utility `Zip_parser` to create a binary schema file that is used by the database engine (the entity that controls access to the database instances). The engine uses the binary schema file at database creation time to preallocate system resources and build symbol and hash tables for query access. Zip_RTDBMS provides another utility `create_database` that can be used to create an empty database instance.

Zip_RTDBMS also provides a frontend function library. The function library provides the numerous functions needed to access the database. The functions provide a low-level call interface to the database engine. These functions are divided into five areas that are briefly described below:

**Connection Management** - Provides the means for creating and destroying database instances, and connecting or disconnecting from a database server.

**Data Management** - Provides query preparation functions along with actual query primitives including those for retrieving, inserting, deleting, and modifying data.

**Set Management** - Provide functions for working with sets of data that may be returned as the result of a data retrieval command.

**Transaction Management** - Provides functions for beginning a transaction and committing or aborting the transaction.

**General Error Management** - Provides a mechanism which can be used to specify a user-defined cleanup function.

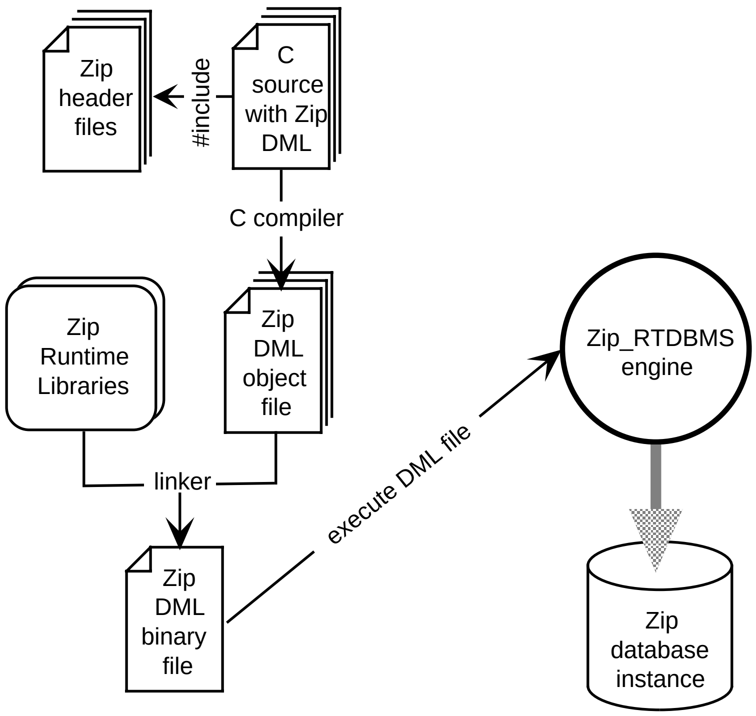


Figure 5.3: Data Manipulation in Zip_RTDBMS

Figure 5.3 shows the process that the user goes through to create a program with Zip_RTDBMS data manipulation commands. The user creates a C program (Zip DML file) that utilizes functions from the frontend library. This file is compiled and linked with the Zip_RTDBMS library to create a program that interacts with the Zip_RTDBMS engine to process queries on the database.

Queries in Zip_RTDBMS are set up in three steps. First, query preparation functions are used to specify the target relation (or table) and attributes (or columns) for the query. Second, a query predicate must be specified. This predicate is used to indicate which tuples (or rows) of the table should actually be operated on by the query. For example, if there is a table of stock information, a query for retrieving data may specify that only the stocks with prices greater than 100 and less than 1000 should have their identifiers and price returned. Here, the predicate is `price > 100 AND price < 1000`. The last step is to actually initiate the desired query (i.e. select, update, insert, delete).

Predicate expressions are somewhat limited in Zip_RTDBMS. Use of the relational operators is limited to comparisons between attributes and values. Thus, it would not be possible to write a predicate which involves the direct comparison of two attributes using Zip_RTDBMS commands. Also, the documentation contains the following statement:

> "To achieve optimum performance at run-time, Zip_RTDBMS requires the designation and participation in a query predicate of a primary key attribute for each relation in the data base."

This implies that a relation must identify a key attribute (the first attribute declared unless otherwise specified), and to achieve optimal performance, the key attribute must appear in the query predicate.

Actual experimentation with the predicates in Zip_RTDBMS has produced results that are not consistent with the documentation. The only relational operator that appears to work is equal (EQ). Also, though not specified in the documentation, only attributes that have an index defined at data definition time seem to work with the EQ operator. Due to these problems, rather than parse the RTSQL predicates to Zip_RTDBMS function calls, all of the data searching will be done by the code built by the RTSQL preprocessor that we have developed.

### 5.1.2 Interface Library

To facilitate the use of Zip_RTDBMS, we developed an interface using three C++ classes. The first class, called `ZipDB_t`, provides a C++ wrapper around the Zip_RTDBMS. The specification of this class is noted in Figure 5.4. The `ZipDB_t` class provides interface functions for the Zip_RTDBMS library functions responsible for data manipulation and connection management. For example, this class provides a function `connect()` that maps directly to the Zip_RTDBMS function `connectDB(DBid_t *masterDBid)`. This is the function that is used to request a connection with the Zip_RTDBMS engine. Note that the interface version, `connect()`, has abstracted the database identifier `masterDBid` from the user of the `ZipDB_t` class. Management of this value and other Zip_RTDBMS variables have been localized to

```
class ZipDB_t {
  public:

    ZipDB_t(char *);
    virtual ~ZipDB_t();

    DBid_t       * masterDBid;
    DBhandle_t   * DBhandle;
    char         * schema_designator;

    // Connection member functions
    int create();
    int destroy();
    int connect();
    int disconnect();
    int describe();

    // Low level query preparation functions
    int declare_rel(relation_t * relation);
    int add_attr(attr_base_t * attr);
    int declare_pred(attr_base_t attr, relop_t relop, pointer_t value);
    int declare_op(logop_t op);

    // DML member functions - use after query preparation functions executed
    int insert_row();
    int update_rows();
    int delete_rows();
    // This select function may return multiple tuples
    int select_rows(TIDset_t *&tids);
    // so provide a counter and an iterator
    int count_tuples(TIDset_t *tids);
    int next_tuple(TIDset_t *&tids);

    // tuple level operations
    int select_row_by(TID_t tid); // called to load target tuple
    int update_row_by(TID_t tid); // called after load target tuple
    int delete_row_by(TID_t tid); // called after target tuple found
};
```

Figure 5.4: Class Definition for Zip_RTDBMS Interface

```
class relation_t {
 public:
    relation_t(const char *);
    virtual ~relation_t() {delete rel_name;}

    string_t   * get_name()     { return rel_name; }
    rel_t        get_type()     { return rel_type; }
    ubyte4_t     get_size()     { return rel_size; }

  protected:
    string_t   * rel_name;
    rel_t        rel_type = BOUNDED; // zip table type
    ubyte4_t     rel_size = 1000;    // maximum number of tuples in table
};


class attr_base_t {
 public:
   attr_base_t(const char *);
   virtual ~attr_base_t() { delete name; }

   string_t * name;
   DataType_t type;

   virtual int add_to_rel(DBhandle_t *) {}
   virtual int add_to_pred(DBhandle_t *, relop_t) {}
};
```

Figure 5.5: Class Definitions Relations and Attributes

the ZipDB_t class.

Two other classes are provided to complete the interface, one for relations (tables), and the other for attributes (columns) (see Figure 5.5). The relation class relation_t is the base class for all relations that are accessed through the interface library. For example, if the user specifies a table with three attributes, the RTSQL preprocessor will create a class for the new table using inheritance, relation_t as the base class, and adding member variables for the attributes. relation_t contains member variables for storing the characteristics of the table that are needed by Zip_RTDBMS. These variables include the name of the relation, the type of the relation, and the maximum size of the relation. Currently, the `relation_t` class supports only BOUNDED relations since they most closely resemble the tables found in SQL.

An attribute base class `attr_base_t` is also defined, to represent the attributes (or columns) a user would place in a relation. As in the `relation_t` class, the `attr_base_t` holds information needed by Zip_RTDBMS, including the name of the attribute and the name of the type of the attribute. The actual value of the attribute must also be stored, but since the type is not known until the user creates a table, the definition of this type is deferred to a derived class which uses `attr_base_t` as its base class. Ideally, this specification would be done using C++ templates, but since these are not available in the version of C++ being used for the implementation, derived classes for the most popular types where included in the interface.

The attribute class also has two member functions which are directly mapped to functions provided by Zip_RTDBMS. The first, `add_to_rel`, is used to specify which attributes will be involved in a query. As mentioned in section 5.1.1, Zip_RTDBMS requires the target relation and attributes of a query be specified before the actual query is issued to the database engine. Assuming that the target relation has already been identified, `add_to_rel` will be used to add the appropriate attributes. For example, if the user does a `SELECT id, price FROM stock`, `stock` is the target relation, and `id` and `price` are the attributes involved in the query. The RTSQL preprocessor takes care of determining this information from the user's query and building the appropriate commands for preparing the query for Zip_RTDBMS.

The second member function in the attribute class is `add_to_pred`. This function is could be used to build the predicate used by Zip_RTDBMS. If the problems experienced with the predicate functionality of Zip_RTDBMS are corrected, then this function will be used to build the query predicates.

## 5.2 RTSQL Preprocessor

The RTSQL preprocessor will translate C++ code containing RTSQL code to C++ code with interface library as discussed in section 5.1.2. As shown in Figure 5.6, the

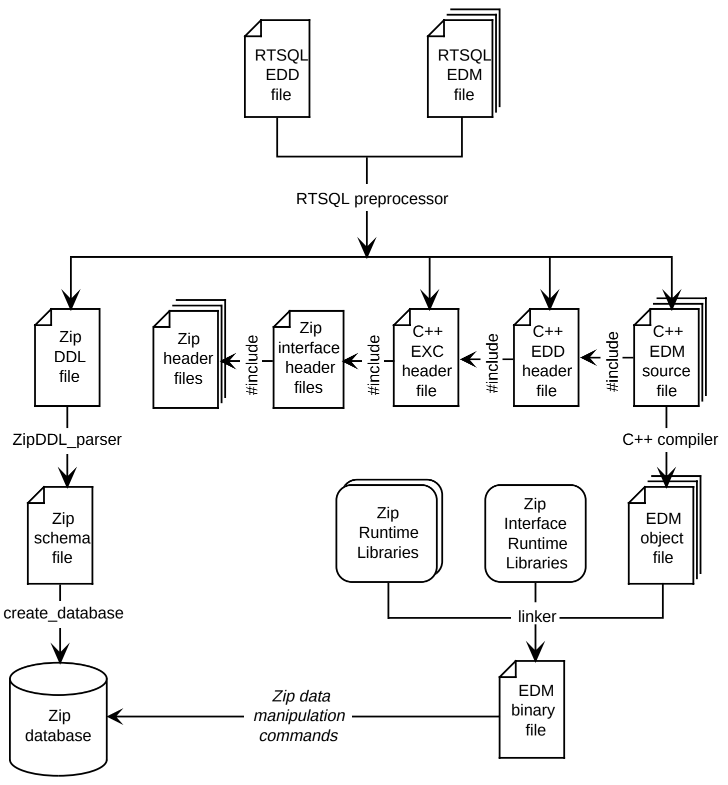


Figure 5.6: RTSQL Preprocessor

user must create two files, one containing data definition statements, the other with RTSQL data manipulation statements embedded in C++ code. A simple example of what these files look like is shown in Figure 5.7. The first file, **tst.EDD**, contains a `CREATE SCHEMA` command and a `CREATE TABLE` command. The `CREATE TABLE` command specifies two attributes (one with a timestamp) and three constraints. The second file, **tst.EDM**, has an `INSERT` statement contained in a loop, and a `SELECT` statement after the loop. Once these two programs are run through the preprocessor and compiled, they can be used to place 100 rows of information in the `stock` table (which is subsequently viewed using the `SELECT` statement.

When these two files (**tst.EDD** and **tst.EDM**) are run through the RTSQL preprocessor `rtsql`, four files are created. Here is a brief description of the resulting files:

**The tst.DDL file.** This file contains Zip_RTDBMS data definition commands. It must be run through the `Zip_parser` command to create a schema file for Zip_RTDBMS.

**The tstEDM.cc file.** This file contains the translation of the **tst.EDM** file from embedded RTSQL to C++ code with library calls to the Zip_RTDBMS interface.

**The tstEDD.h file.** This file contains the class definition corresponding to the `CREATE TABLE` command in the **tst.EDD** file. These classes are derived from the base classes for tables and attributes that are contained in the Zip_RTDBMS interface. Note that this filename appears in an `#include` statement in **tstEDM.cc**.

**The tstEXC.h file.** This file contains classes corresponding to the RTSQL blocks specified in **tst.EDM**. Each class stores the condition handlers and timing constraints associated with a block. This file also includes the Zip_RTDBMS interface header file, which in turn includes the header files from Zip_RTDBMS. This file is in an `#include` statement of **tstEDD.h**.

```
/******************************************************************
** File tst.EDD
** This file contains the RTSQL data definition commands
*/
/* stock table */
EXEC SQL CREATE TABLE stock (
id INTEGER,
price   REAL WITH TIMESTAMP,
change  REAL,
CONSTRAINT valid_data CHECK (id > 0 AND id <= 99 AND price <= 1000.0)
CONSTRAINT pos_change CHECK (change > 0 OR change = 0)
CONSTRAINT price_avi CHECK
  (price > 0 AND TIMESTAMP(price) > CURRENT_TIMESTAMP - INTERVAL '5' SECOND)
);

/******************************************************************
** File tst.EDM
** Thid file contains the embedded RTSQL data manipulation commands
*/
#include "tstEDD.h"
main() {
  int i;
  float j;
  for (i=1; i<=100; i++) {
     j = i / 2.0;
     EXEC SQL INSERT INTO stock ( id, price ) VALUES ( :i, :j );
  }
  EXEC SQL SELECT * FROM stock;
  printf("That's all folks...\n");
}
```

Figure 5.7: Example of RTSQL Data Definition and Data Manipulation Files

### 5.2.1 Program Structure

An SQL preprocessor developed in [LMB95] was used as a basis for the RTSQL preprocessor. This SQL preprocessor was based upon SQL-89, a predecessor of SQL2. Though this preprocessor produces a simple call to a nonexistent C-routine `exec_sql()` for each SQL statement, it provided a good foundation for the RTSQL preprocessor.

The SQL preprocessor was implemented using the compiler utilities *flex* and *bison*. *flex* is a tool developed by the GNU Project of the Free Software Foundation for automatically generating lexical analyzers. It is based upon an earlier version called LEX [Les75]. In the SQL preprocessor, the lexical analyzer is responsible for scanning the input files and identifying the SQL tokens. This requires the lexical analyzer to run in two modes, one that simply passes C++ code to an output file, the other that passes SQL tokens to the parser generated by *bison*. *bison* is an automatic parser generator also developed by the GNU Project. It is based upon an earlier version called YACC [Joh75]. In the SQL preprocessor, the parser buffers the SQL tokens which constitute an SQL statement. The essential information is then passed to the non-existent C-routine `exec_sql()`, which could be implemented to interact with an existing database system.

Initially, the SQL preprocessor was extended to create the RTSQL preprocessor by modifying the input files for flex and bison. The input file for flex, which identifies the valid tokens of the language, was extended with the new keywords specific to RTSQL. The input file for bison, which contained the grammar rules for SQL was replaced with a grammar file specific to RTSQL. A copy of this grammar can be found in Appendix A.

Four areas needed to be developed in creating the RTSQL preprocessor. Initially, the data definition statement `CREATE TABLE` was implemented. This included adding timestamps to data and a constraint checking mechanism for both logical constraints and temporal constraints on the data. Next, the basic data manipulation operations `SELECT`, `INSERT`, `UPDATE`, and `DELETE` were implemented. The condition handling

```
/**********************************************
** Example RTSQL data definition file
*/
EXEC SQL CREATE SCHEMA stock;

EXEC SQL CREATE TABLE stock (
id INTEGER,
price   REAL WITH TIMESTAMP,
name CHAR(10)
);
```

Figure 5.8: RTSQL Data Definition File

mechanism was then added to react to constraint violations. Finally, timing constraints on actions were added. Each of these areas is discussed in the following sections.

### 5.2.2 Data Definition Operations

Recall that all of the data definition statements are placed in a single file that will represent the database schema. This file may contain any number of `CREATE TABLE` commands. When this file is run through the RTSQL preprocessor, two files are created. First, a data definition file suitable for the ZipDDL_parser is created. Second, a relation class is derived from the `relation_t` class with the addition of the attributes specified in the `CREATE TABLE` statement. If it has been specified that an attribute should have a timestamp maintained (by the `WITH TIMESTAMP` clause), the preprocessor will automatically generate another attribute for storing the timestamp value.

Figures 5.8, 5.9, and 5.10 are an example of an RTSQL data definition file and the files resulting from the preprocessor. Note that one of the attributes (`price`) had a timestamp specified. This results in the creation of an additional column in the relation for the Zip_RTDBMS database. Also, the interface class `stock_t` derived from `relation_t` has a separate attribute for the timestamp value.

The table definition may also contain constraint predicates. Note that constraints

```
/**********************************************
** Resulting data definition file for Zip_RTDBMS
*/
#include "zip.h"
/*
** Filename: ex1.DDL
*/

/*
** Database characteristics
*/
define database db_name          ("stock");
define database db_dir_path      ("/tmp");
define database db_phys_addr     (0x0);
define database db_size          (400000);
define database db_grants        (0777);

/*
** Table stock
*/
create table stock (
  id          byte4,
  price       byte8f,
  price_ts    tstamp,
  name        string[10]);

define table stock (BOUNDED, 1000);
define index stock_idx on stock (id) load(50%), distinct(500);
```

Figure 5.9: RTSQL Output File for Zip_RTDBMS Schema

```
/**********************************************
** Resulting data definition file for interface
*/
#include "ex1EXC.h"

class stock_t : public relation_t {
public:
  stock_t(char * name) : relation_t(name),
    id("id"), price("price"), price_ts("price_ts"), name("name",10) {}
  attr_byte4_t  id;
  attr_byte8f_t  price;
  attr_tstamp_t  price_ts;
  attr_string_t  name;
};

stock_t stock("stock");
TIDset_t * tids;
TID_t prevTID, currTID;
Zip_rtdbms  DB("/tmp/stock.schema");
```

Figure 5.10: RTSQL Output File for Interface

are handled by the code built by the RTSQL preprocessor, not Zip_RTDBMS. These constraint predicates are divided into two categories, *logical* constraints and *temporal* constraints. A constraint will be identified as a temporal constraint if it involves any time expressions. The time expression usually references the timestamp value of one of the attributes. The division of logical and temporal constraints is done by the preprocessor so that two constraint checking functions can be created. This is due to the nature of when these constraints are checked. Logical constraints are generally checked when data is written. As mentioned in section 4.1.1, temporal constraints may be checked actively (using timers and alarms) or passively (when a data item is read). This implementation has chosen passive checking since it provides some context for the constraint violations to be handled. The `check_constraint` function will be called whenever an `INSERT` or `UPDATE` operation is performed. The `check_time_constraint` function will be called whenever a `SELECT` operation is performed.

Figures 5.11 and 5.12 show examples of constraint specifications for logical and

```
/*
** Data constraints for logical consistency
*/
CONSTRAINT valid_data CHECK (id > 0 AND id <= 99 AND price <= 1000.0)
CONSTRAINT pos_change CHECK (change > 0 OR change = 0)

/*
** Resulting check_constraint function - pseudo code
*/
int check_constraints(byte4_t *id, byte8f_t *price, byte8f_t *change) {
  int ans[10];

  ans[0] = 1; /*Assume TRUE*/
  if (id != NULL)
    ans[0] = ans[0] && *id <= 99;
  if (id != NULL)
    ans[0] = ans[0] && *id > 0;
  ans[1] = 1; /*Assume TRUE*/
  if (price != NULL)
    ans[1] = ans[1] && *price <= 1000.0;
  ans[1] = ans[1] && ans[0];
  if (!(ans[1])) {
    printf("valid_data constraint violated\n");
    /* Set up appropriate handler */
  }

  ans[0] = 1; /*Assume TRUE*/
  if (change != NULL)
    ans[0] = *change == 0 || *change > 0;
  if (!(ans[0])) {
    printf("pos_change constraint violated\n");
    /* Set up appropriate handler */
  }

  if (problem found) {
    /* raise condition */
  }
} /* end check_constraints */
```

Figure 5.11: check_constraint Function Created by RTSQL Preprocessor

```
/*
** Data constraint for temporal consistency
*/
CONSTRAINT price_avi CHECK
  (price > 0 AND TIMESTAMP(price) > CURRENT_TIMESTAMP - INTERVAL '5' SECOND)

/*
** Resulting check_time_constraint function - pseudo code
*/
int check_time_constraints(byte4_t *id, byte8f_t *price, tstamp_t *price_ts,
                           byte8f_t *change) {

  ans[0] = 1; /*Assume TRUE*/
  if (price_ts != NULL)
    ans[0] = ans[0] && *price_ts > current_sec() - 5;
  if (price != NULL)
    ans[0] = ans[0] && *price > 0;
  if (!(ans[0])) {
    printf("price_avi constraint violated\n");
    /* Set up appropriate handler */
  }

  if (problem found) {
    /* raise condition */
  }
} /* end check_time_constraints */
```

Figure 5.12: `check_time_constraint` function created by RTSQL preprocessor

| *AND* | T | F | U |
|---|---|---|---|
| T | T | F | U |
| F | F | F | F |
| U | U | F | U |

| *OR* | T | F | U |
|---|---|---|---|
| T | T | T | T |
| F | T | F | U |
| U | T | U | U |

| *NOT* | T | F | U |
|---|---|---|---|
| | F | T | U |

Figure 5.13: Three-Valued Logic Tables for SQL

temporal consistency respectively (they are shown in context in Figure 5.7). The resulting functions for the logical constraints (`check_constraints`) and temporal constraints (`check_timing_constraints`) are shown. These functions are parameterized with pointers to all the possible attribute values. In the case of `check_constraints`, this is all of the attributes specified in the `CREATE TABLE` definition. In the case of `check_timing_constraints`, it is all of the attributes along with their timestamps. Pointers to these values are used so that if a value is not defined at the point at which the constraint function is called, a NULL pointer will be sent. SQL uses three valued logic: TRUE, FALSE, UNKNOWN. Thus, if a simple boolean expression involves a value which is not defined, the expression evaluates to UNKNOWN. Figure 5.13 shows the truth tables for the SQL three valued logic.

For each constraint specified, a series of if statements are constructed to reflect the semantic behavior of the three valued logic. Recall that in SQL, a constraint is violated only if it evaluates to FALSE. The series of if statements corresponding to a given constraint are designed to return TRUE where the three valued logic would evaluate to either TRUE or UNKNOWN.

Within the constraint checking functions, a number of constraints are checked. It is possible that more than one constraint violation could be detected within the function. For example, if one constraint specifies that the pressure value must be greater than zero, and another specifies that the temperature value is less than 500, then both constraints will be violated if an attempt is made to insert a pressure value of -1 and a temperature value of 625. In this case, the SQL standard specifies that the system will recognize one of the constraint violations as the "primary" violation, while any other violation will be placed in a diagnostics area, space permitting. The standard does not specify any precedence ordering of constraint violations. This implementation behaves similarly, in that only one exception will be raised upon detection of a constraint violation. At this time, the implementation only keeps track of one of the violations (no diagnostic area has been provided), though this could be extended in future versions.

When the function detects a constraint violation, a unix signal will be raised. This signal will have a signal handling function associated with it which corresponds to the user's condition handling routine if it exists. Section 5.2.4 will discuss in detail how this condition handling is done.

### 5.2.3 Basic Data Manipulation Operations

After the data definition commands were implemented, the next step was to implement the basic database operations. This involved associating the appropriate Zip_RTDBMS interface library calls with the corresponding data manipulation operations. As mentioned in section 5.1.1, a query must be setup before the actual query function is issued to Zip_RTDBMS. The preprocessor builds the appropriate function calls to set up the query, and issue the actual database operation.

Since the Zip_RTDBMS predicates are not being utilized at this time, any searching of the data is done within the code built by the RTSQL preprocessor. The `WHERE` clause is used indicate the searching criteria in the `SELECT`, `DELETE`, and `UPDATE` statements. Code is built to extract the entire table from the database, which will then be searched using this criteria. In the case of an `UPDATE` or `DELETE`, when an appropriate tuple is located using the search criteria, the actual operation is then performed on that tuple in the Zip_RTDBMS.

Figure 5.14 shows an example of the code produced by the RTSQL preprocessor in translating an `INSERT` statement and a `SELECT` statement. Note the query preparation functions that appear before the actual query is issued.

### 5.2.4 Condition Handling

The condition handling mechanism in RTSQL is specified within the block structure of SQL/PSM. The blocks provide a context for the condition handlers specified by the

```
/*******************************************************
** INSERT INTO stock ( id, price ) VALUES ( :k, :j );
*/
stock.id.value = k;
stock.price.value = j;
if (stock.check_constraints(&(stock.id.value), &(stock.price.value))) {
  DB.reset_query_params();
  DB.declare_rel(&stock);
  DB.add_attr(&(stock.id));
  DB.add_attr(&(stock.price));
  DB.add_attr(&(stock.price_ts));
  stock.price_ts.value = current_sec();
  DB.insert_row();
} /* if check_constraint */


/*******************************************************
** SELECT id, price FROM stock WHERE id > 4 OR price = 1.0;
*/
DB.reset_query_params();
DB.declare_rel(&stock);
DB.add_attr(&(stock.id));
DB.add_attr(&(stock.price));
DB.select_rows(tids);
while (DB.next_tuple(tids))
  if (stock.id.value > 4 || stock.price.value == 1.0)
    if (stock.check_time_constraints(&(stock.id.value),
                                     &(stock.price.value), NULL))
      printf("  %d  %f\n" , stock.id.value, stock.price.value);
```

Figure 5.14: Code Produced by Preprocesor for `INSERT` and `SELECT`

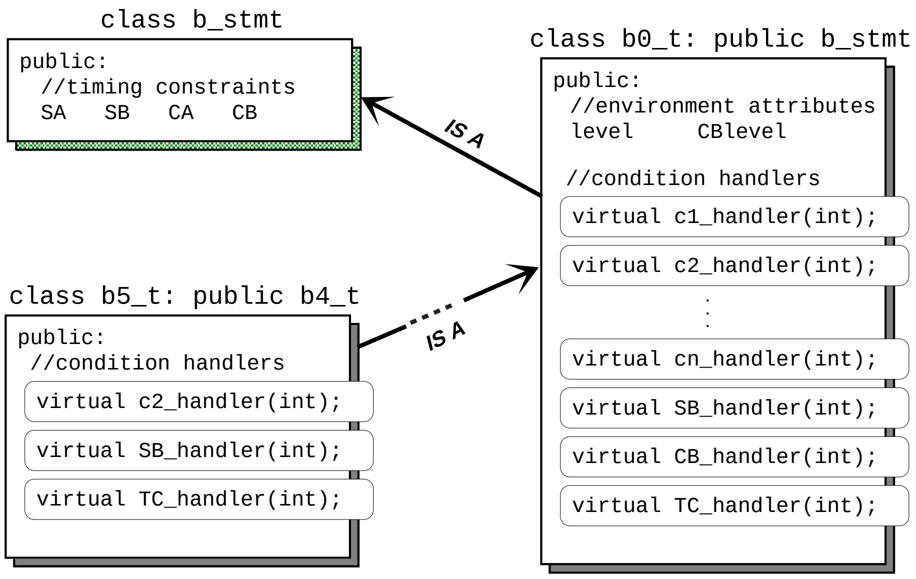


Figure 5.15: Class Definitions for Statements and Blocks

user. Note that blocks may be nested, thus a condition handler of an outer block may be used in an inner block unless redefined by the inner block. The implementation must keep track of the valid handlers for each block. Also, a set of default handlers must be provided.

To associate handlers with blocks, blocks are defined using C++ classes with the handlers as methods in the class. As shown in Figure 5.15, the C++ inheritance mechanism is used to define blocks. Note that the base class `b_stmt` is for statements in general, all of which may have timing constraints specified (see section 4.1.1). Variables are provided to store the actual values of each of the four timing constraints: `SA` for `START AFTER`, `SB` for `START BEFORE`, `CA` for `COMPLETE AFTER`, and `CB` for `COMPLETE BEFORE`. Also, the class provides variables for keeping track of the initial block which should be searched for an appropriate handler. The next class, `b0_t` is the base class for all blocks, and as such, will define the default handlers for all possible constraint violations. These default handlers represent unhandled constraint violations, and will cause program termination with an appropriate error message.

When a block is declared in the RTSQL code, the preprocessor will assign the

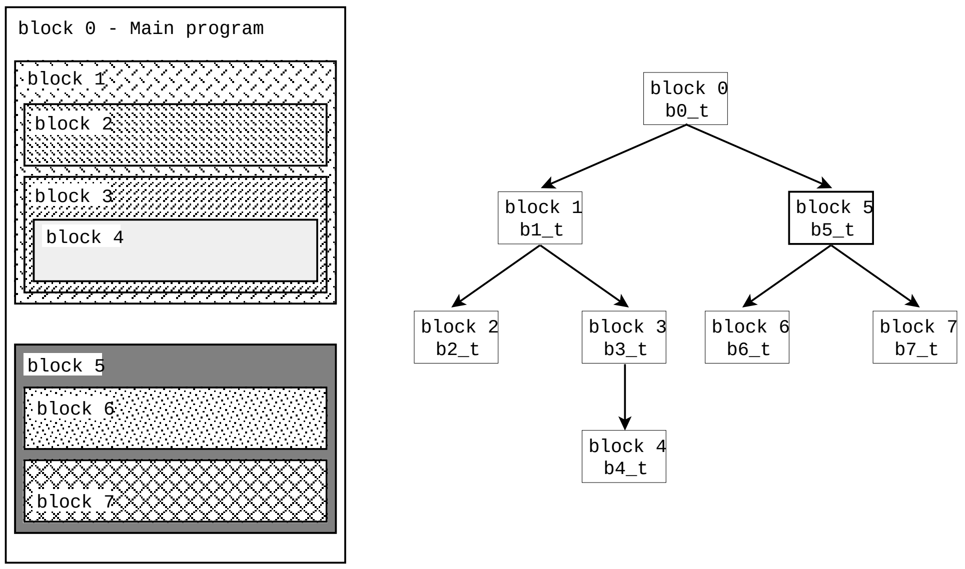


Figure 5.16: Class Hierarchy for Blocks

block a number and define a class corresponding to this block. In the case where it is an outermost block, it will be derived from the base class for all blocks `b0_t`. In the case where it is nested within another block, it will be derived from the class for that immediate outer block. Figure 5.16 shows a series of block declarations, and the corresponding inheritance tree built by the preprocessor. The block numberings shown would be generated by the preprocessor as it parses through the code. For example, since block 3 is immediately nested with block 1, the class corresponding to block 3 will use the class for block 1 as its base class. This methodology allows the inheritance mechanism of C++ to manage the association of handlers with the blocks, especially in the case where blocks are nested. For example, it a `check_id` constraint handler has been defined in block 1 but not defined in block 3, the class corresponding to block 3 will inherit the `check_id` handler from block 1. This reflects the desired semantics of having an outer block's handler declarations available for the inner blocks. It also provides for the default handlers, since if a handler is never specified in a block or any of its ancestors, it will inherit the default handler from the base class `b0_t`.

The actual code for the handlers is rather simple. The user will specify a routine

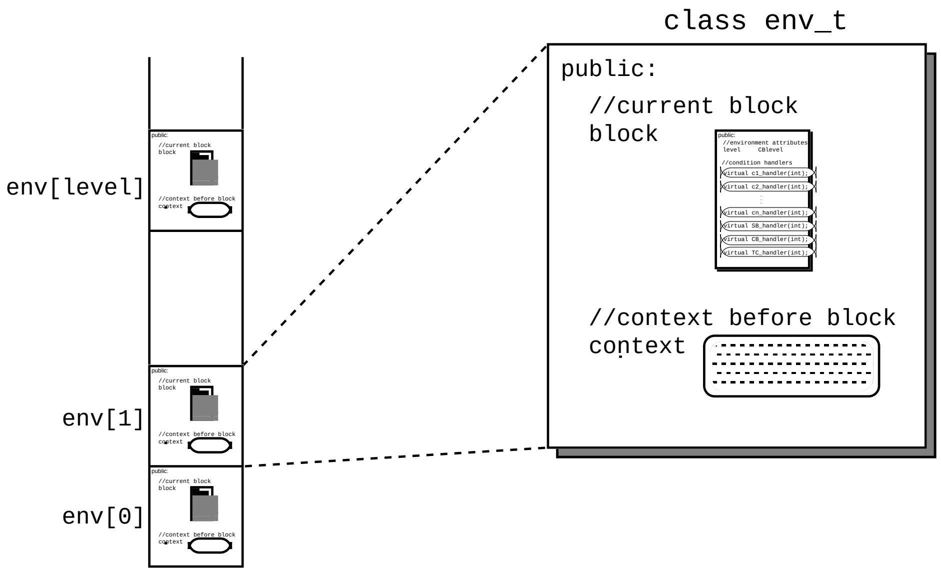
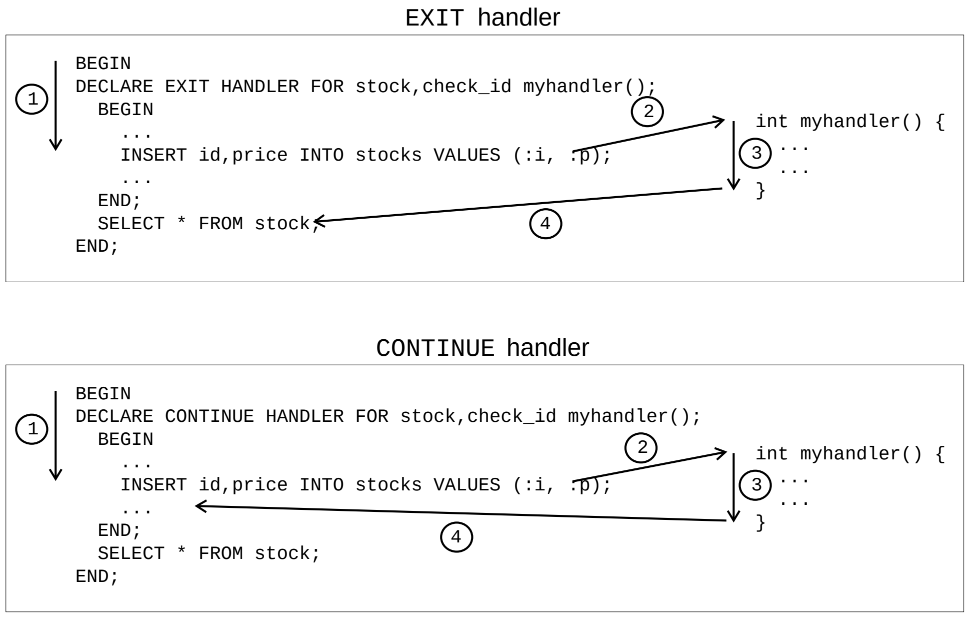


Figure 5.17: Environment Stack

which is to be used to handle the condition. A call to this routine will be placed in the condition handler. Also, the user may specify that the handler is a `CONTINUE` handler or an `EXIT` handler. Figure 5.18 shows the flow of control for both types of handlers. If it is a `CONTINUE` handler, then flow of control will simply return to the statement following the one that raised the constraint violation. If an `EXIT` handler is specified, then the condition handler should return control to the the statement following the block in which the condition occurred (as opposed to the block containing the handler). Note that this will simply cause the remainder of the block to be abandoned. To handle both of these possibilities, the preprocessor maintains a stack for storing information about the programming environment. This stack `env` (see Figure 5.17 is designed to save information about the execution environment at certain points in the program, so that if a block must be abandoned, there will be some previous context for the program execution to return to. Note that each element of the stack can store an environment state `context` and a block.

Before a block begins execution, the current environment and the block will be stored on the top of the stack. If during the execution of the block, a constraint violation occurs, and the handler is an `EXIT` handler, the environment at the top



```
BEGIN
DECLARE EXIT HANDLER FOR stock,check_id myhandler();
  BEGIN
    ...
    INSERT id,price INTO stocks VALUES (:i, :p);
    ...
  END;
  SELECT * FROM stock;
END;

int myhandler() {
  ...
  ...
}
```

```
BEGIN
DECLARE CONTINUE HANDLER FOR stock,check_id myhandler();
  BEGIN
    ...
    INSERT id,price INTO stocks VALUES (:i, :p);
    ...
  END;
  SELECT * FROM stock;
END;

int myhandler() {
  ...
  ...
}
```

Figure 5.18: Flow of Control for EXIT and CONTINUE Handlers

of the stack will be restored, the top element of the `env` stack will be removed, and execution will continue at the statement following the block which caused the constraint violation. If the block has no constraint violations, the the top element of the `env` stack will simply be removed, causing the previous environment and block to return to the top of the stack.

Before a statement begins execution, it is also saved to the top of the stack, without a copy of the current environment. The environments are saved only for blocks, as the semantics of EXIT handlers are defined only within the context of a block.

### 5.2.5 Timing Constraints on Actions

Four timing constraint clauses on actions were implemented: `START BEFORE`, `START AFTER`, `COMPLETE BEFORE`, and `COMPLETE AFTER`. These constraint clauses are specified at the end of a statement or block, and will be translated into a series of statements that will appear before and after the actual translation of the statement.

These statements include calls to routines for setting alarms and calls to routines for delaying execution. The following section discusses some of the implementation issues for timing constraints. The next sections describe the methodology for each of the constraints implemented including how they were interleaved in the code appearing before and after the actual statement (the precode and the postcode).

### Implementation Issues

Before a statement or block begins execution, all of the time expressions contained in the timing constraint clauses must be evaluated, producing an absolute time value for each clause. This absolute time value is expressed in seconds and microseconds since 00:00 Universal Coordinated Time, January 1, 1970. The time expressions often involve references to datetime valued functions such as `CURRENT_TIMESTAMP`. Recall that within a statement or block, all of the calls to these datetime valued functions should appear to have been evaluated at the same instance of time. Thus, if two of the clauses have the same time expression, they will evaluate to the same absolute time value.

The preprocessor will translate the time expressions in RTSQL into C++ expressions which will be evaluated when the code is executed. To facilitate the translation of these expressions, a C++ class called `my_time_t` was developed to support manipulation of time values. The class `my_time_t` supports time values consisting of a seconds field and a microseconds field. This class also provides addition and subtraction operators for the time values as well as the full set of comparison operators (i.e. less than, less than or equal to). Also provided is a function called `current_ts()`, which can be used to determine the current timestamp, and will return time values of type `my_time_t`.

After all of the timing constraint expressions are evaluated, they must be checked for consistency. A function called `check_consistency` was developed for this purpose. This function compares the values of constraints which would be in obvious conflict. Three cases are checked as is shown in Figure 5.19. In case A, the time value of the `START BEFORE` clause should be earlier than the time value of the `START AFTER`

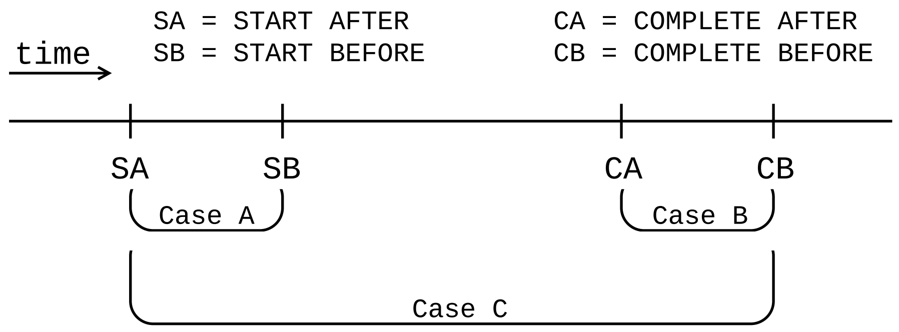


Figure 5.19: Three Cases Checked by `check_consistency` Function

clause. In case B, the time value of the `COMPLETE BEFORE` clause should be earlier than the time value of the `COMPLETE AFTER` clause. In case C, the time value of the `START AFTER` clause should be earlier than the time value of the `COMPLETE BEFORE` clause. Each of these cases would definitely cause a constraint violation if they were not true, and so a condition value `TC_CONSTRAINT` will be raised to indicate that one of these timing constraint conflicts has occurred. For this consistency check to work properly in situations where only some of the constraints may have been specified, default values for the various clauses are provided. `START AFTER` and `COMPLETE AFTER` clauses use a default value of 0, which corresponds to January 1, 1970. The `START BEFORE` and `COMPLETE BEFORE` clauses use a default value of 2147483647, the maximum integer value INT_MAX. This corresponds to the date January 18, 2038.

The `check_consistency` function also checks the time value of the `COMPLETE BEFORE` clause against the time value of the `COMPLETE BEFORE` clause of the next outer block. For example, suppose we have the following:

```
BEGIN
  SELECT * FROM stocks
    COMPLETE BEFORE CURRENT_TIMESTAMP + INTERVAL '5' SECOND;
  ...
END COMPLETE BEFORE CURRENT_TIMESTAMP + INTERVAL '10' SECOND;
```

In this case, the deadline (e.g. time value of the `COMPLETE BEFORE` clause) for the `SELECT` statement is earlier than the deadline for the outer block. But suppose we had the following:

```
BEGIN
  SELECT * FROM stocks
    COMPLETE BEFORE CURRENT_TIMESTAMP + INTERVAL '20' SECOND;
  ...
END COMPLETE BEFORE CURRENT_TIMESTAMP + INTERVAL '10' SECOND;
```

In this case, the deadline of the SELECT statement is later than the deadline of the next outer block. Thus, the deadline on the block will expire before the deadline on the SELECT statement, and a timing constraint violation would occur on the whole block instead of the SELECT statement. This means that the handler used for the timing constraint violation should be the one intended for the block, not the statement. To handle this situation, the deadline of the current statement or block will be be checked against the deadline of the next outer block. If the statement or block has a later deadline than the next outer block, it will be reset to match the earlier value of the outer block. Since this check may reset the value of the COMPLETE BEFORE clause, it must be done before the consistency check of the timing constraints is performed.

### Implementation of Timing Constraint Clauses

The timing constraint clauses can be divided into two categories. The first category, *delay constraints*, contains the clauses that may cause a delay in execution. The second category, *deadline constraints*, contains statement which specify some deadline which must be met. Each of these categories is described in the following paragraphs.

**Delay constraints.** This category includes the START AFTER clause and the COMPLETE AFTER clause. The preprocessor will translate these clauses into statements which include a call to a routine which suspends execution until a certain time interval has elapsed. For example, suppose we have the following statement:

```
SELECT id, price FROM stocks WHERE price > 1000.0
  START AFTER CURRENT_TIMESTAMP + INTERVAL '5' SECOND;
```

The START AFTER clause indicates that the statement should begin execution after the current timestamp plus 5 seconds. This particular constraint is used to delay the time at which this statement begins execution. If the system is ready to execute this statement before this time, it must wait until this time has elapsed.

Let SA be the time value of the `START AFTER` clause, let `CA` be the time value of the `COMPLETE AFTER` clause, and let `current_ts()` be the actual current time. The `START AFTER` clause will be translated by the preprocessor into the following actions (psuedo code shown):

```
if SA > current_ts() then
  delay (SA - current_ts())
```

The `COMPLETE AFTER` clause will be translated by the preprocessor into the following actions (psuedo code shown):

```
if CA > current_ts() then
  delay (CA - current_ts())
```

**Deadline constraints.** This category includes the `START BEFORE` clause and the `COMPLETE BEFORE` clause. The preprocessor will translate the `START BEFORE` clause into a check to see if the constraint has been violated when execution of the statement is about to begin. For example, suppose we have the following statement:

```
SELECT price FROM stocks WHERE price < 1000.0
  START BEFORE CURRENT_TIMESTAMP + INTERVAL '5' SECOND;
```

The `START BEFORE` clause indicates that the statement should begin execution before the current timestamp for the block plus 5 seconds. If this statement begins execution after this time the `START BEFORE` constraint is violated, and the condition `SB_CONSTRAINT` is raised.

The `COMPLETE BEFORE` clause will be translated into statements which may include a call to a routine which will set a timer expiration time. When the timer expires, a signal will be sent to the calling routine indicating that the timer has expired. For example, suppose we have the following statement:

```
SELECT id, price FROM stocks WHERE price > 1000.0
  COMPLETE BEFORE CURRENT_TIMESTAMP + INTERVAL '5' SECOND;
```

The COMPLETE BEFORE clause indicates that the statement should finish execution before the current timestamp for the block plus 5 seconds. If this statement finishes execution after this time the COMPLETE BEFORE constraint is violated, and the condition CB_CONSTRAINT is raised.

Let SB be the time value of the START BEFORE clause, let CB be the time value of the COMPLETE BEFORE clause, let current_ts() be the actual current time, and let set_alarm() and unset_alarm() be routines for setting and unsetting a timer. The START BEFORE clause will be translated by the preprocessor into the following actions (psuedo code shown):

```
if SB < current_ts() then
  set up handler routine for SB_CONSTRAINT
  raise SB_CONSTRAINT
```

The COMPLETE BEFORE clause will be translated by the preprocessor into the following actions (psuedo code shown):

```
set up handler routine for CB_CONSTRAINT
set_alarm() for CB
** execute statement **
unset_alarm() for CB
```

### Placement of Timing Constraint Code

The previous section described the translation of each of the individual timing constraints. This section will describe the location of the code for the constraints relative to the code representing the translation of the actual statement or block.

Figure 5.20 shows the placement of the code for each of the timing constraints. After the function call to check_consistency(), the code for the START AFTER clause will appear. Recall that the check_consistency() function will have already made sure that the time value of the START AFTER clause is earlier than the COMPLETE BEFORE clause. Next, is the code for setting up the handler for the CB_CONSTRAINT and setting the timer for the COMPLETE BEFORE clause. Note that this timer cannot be set before the code for the START AFTER clause which involves a delay. This is due to the fact the the routines used to cause a delay in the code execution use the same

```
/* Check for timing constraint consistency */
check_consistency()

/* Check time value of START AFTER clause */
if SA > current_ts() then
  delay (SA - current_ts())

/* Set up CB_CONSTRAINT handler and set timer for COMPLETE BEFORE clause */
set up handler routine for CB_CONSTRAINT
set_alarm() for CB

/* Execute the statement or block */
** statement or block code **

/* unset timer for COMPLETE BEFORE clause */
unset_alarm() for CB

/* Check time value of the COMPLETE AFTER clause */
if CA > current_ts() then
  delay (CA - current_ts())

/* Reset alarm for COMPLETE BEFORE clause */
set_alarm() for CB

/* Cleanup code for statement or block */
** cleanup code **

/* unset timer for COMPLETE BEFORE clause */
unset_alarm() for CB
```

Figure 5.20: Placement of Timing Constraint Code

signal as the timer. If the timer is set, and then the delay is initiated, the original timer value will be lost.

The code for the actual statement or block appears next (Figure 5.20). During the execution of this code, the timer could expire, and control would then be transferred to the appropriate `CB_CONSTRAINT` handler. Also, this code may generate a data error condition and pass control to the appropriate condition handler. Even if the timer expires while executing the condition handler for the data error, control would be passed to the same `CB_CONSTRAINT` handler.

Figure 5.20 also shows that upon completion of the block or statement, the timer for the `COMPLETE BEFORE` clause is unset. This is due to the fact that the `COMPLETE AFTER` constraint must be checked, and may involve another call to the routines to delay execution. After this code, the timer for the `COMPLETE BEFORE` clause is reset. Some cleanup code for the statement or block appears next, and then the timer for the `COMPLETE BEFORE` clause is unset as the last action of the statement or block. If the block or statement is nested within another, the timer for the `COMPLETE BEFORE` clause of the outer block is reset.

## 5.3 Discussion

A subset of RTSQL constructs was implemented to demonstrate the feasibility of these extensions to SQL. The implementation allows users to specify RTSQL queries on a simple database system. The focus of the implementation is on data timing constraints, timing constraints on statements, and condition handlers for these constraints if they are violated. There were three reasons why these particular constructs were chosen. First, these constructs represent the least controversial extensions to the SQL language. They require only slight modification of some of the existing constructs of SQL, with the rest of the modifications appearing as pure extensions. For example, providing the construct to allow a user to specify that a data value should have a timestamp is a pure extension to SQL. But allowing the use of time valued functions such as `CURRENT_TIMESTAMP` in constraint specifications requires modifying the current version of SQL.

Second, these constructs are essential in real-time applications. The ability to time constrain data and time constrain actions is one of the most important features of a real-time database. Non real-time applications may also find some of these features useful. Members of the X3 standards committee have commented that many commercial users have expressed interest on placing deadlines on actions in the database system. Third, development of these constructs provides a basis for further research. This implementation is based upon the passive approach of maintaining temporal consistency of data. But an exception handling mechanism is now in place, and could be extended to support the active approach.

One lesson learned from the implementation was the importance of operating system support for timers and signals. The availability of a timer based upon absolute time greatly simplified management of the `COMPLETE BEFORE` constraint in a nested environment. Signals were crucial in providing a means of implementing condition handling. They provided a mechanism to transfer control to routines supplied by the user when a problem was detected. Also, the implementation provided insight to why the semantics of the `EXIT` handlers and `CONTINUE` handlers are defined as they are. `CONTINUE` handlers are implemented by simply using the signal mechanism to transfer control to a handler, and allowing the handler to return to the point at which the signal was raised. `EXIT` handlers are implemented by saving the program context before a block begins execution, so that the handler can return to that environment when it is complete, having abandoned the remainder of the block.

Three areas should be addressed in future work on the implementation. First, the implementation only supports simple database queries. Usually a database system allows a user to create complex nested statements for manipulating data. Support for nested statements in this implementation would distract from the issues being addressed. It is also a feature that is very complex and encompasses a major area of research called query optimization. On the other hand, nested queries would provide a means of exploring condition handling and timing constraints in nested environments. Since this is such an important issue, nesting will be supported for compound statements. This allows us to focus on the concerns of constraints in nested environments without the overhead of complicated queries.

The second area that should be addressed in future work on the implementation is RTSQL transactions. To support transactions in a database system requires the existence of some type of transaction manager. The transaction manager schedules all of the transactions in the system while preserving the consistency of the database. The Zip_RTDBMS data manager was used as a basis for this implementation, and does not provide a transaction manager. Implementation of a transaction manager for our prototype system was beyond the scope of this work. Thus, the implementation assumes there is no contention for resources, and that all transactions commit.

Transactions are actually a controversial issue within the standards committee. The extensions proposed here go way beyond the concepts currently under consideration by the standards committee. Members of this committee have suggested that the constructs we have proposed be introduced at a later time.

A third area that should be addresses in future work on the implementation is directives. Many of the directives provide information to the underlying system so that data access times are predictable and the transaction manager can determine optimal transaction schedules. Addition of these types of features usually requires access to the most primitive operations in the system. In our case, Zip_RTDBMS does provide main memory storage of all data, and supports bounded tables. At this time, we do not provide access to these features, but they could be added at a later time. Also note that our implementation does not support a transaction manager, so many of the directives related to this cannot be implemented at this time.

# Chapter 6

# Evaluation

## 6.1 Implementation Tests

A series of tests were designed to evaluate the implementation. The test suite was incremental in nature, starting with tests that focused on the basic operations implemented and concluding with tests that examined more complicated scenarios such as timing constraint violations in nested blocks. The tests are designed to verify that the constructs implemented exhibit behavior that is consistent with the semantics specified. In the case of the basic operations, this meant verifying that the data definition commands and the data manipulation commands were able to interact with Zip_RTDBMS through the interface we developed. For constraints, this meant verifying that the constraint functions were properly created and executed at the appropriate time. Also, when a constraint violation was detected, that the appropriate handler was initiated. For timing constraints on actions, this meant verifying that the delay constraints actually delayed execution, and that the deadline constraints were properly detected and appropriate handlers executed when violations did occur. These tests are summarized in the following paragraphs.

### 6.1.1 Test Descriptions

**Test 1 - Data Definition Commands.** This test verifies that the tables specified by the user are properly translated into code used by the ZipDDL_parser and objects used in the interface to Zip_RTDBMS. This includes verifying correct creation of both the timestamp columns when an attribute has a timestamp specified, and the constraint functions for the logical and temporal data constraints.

**Test 2 - Data Manipulation Commands.** This test verifies that the interface to Zip_RTDBMS behaves properly. The four basic operations are tested: `INSERT`, `SELECT`, `DELETE`, and `UPDATE`. `WHERE` clauses are also tested on the appropriate statements.

**Test 3 - Logical Data Constraint Violations.** This test verifies that logical data constraints are properly detected and handled by the system. An attempt is made to insert data which will violate the constraints.

**Test 4 - Temporal Data Constraint Violations.** This test verifies that temporal data constraints are properly detected and handled by the system. An attempt is made to read data which is known to be temporally inconsistent.

**Test 5 - Handlers in Nested Blocks.** This test verifies that handlers specified in outer blocks will be invoked in cases where there is no handler in an inner block that contains the constraint violation. Even if the handler in the outer block is an `EXIT` handler, it should exit from the inner block only.

**Test 6 - Basic Timing Constraint Violations** This test verifies that the three types of timing constraint violations are properly detected and handled. These constraints include the `SB_CONSTRAINT` (for the `START BEFORE` clause), the `CB_CONSTRAINT` (for the `COMPLETE BEFORE` clause), and the `TC_CONSTRAINT` (for timing constraint conflicts). The other two timing constraint clauses, `START AFTER` and `COMPLETE AFTER` will also be verified.

**Test 7 - Timing Constraint Violations in Nested Blocks.** This test verifies that handlers specified in outer blocks will be invoked in cases where there is no

| Test# | Passed | Not Passed - Comments |
|---|---|---|
| 1 | X | |
| 2 | X | |
| 3 | X | |
| 4 | X | |
| 5 | X | |
| 6 | X | |
| 7 | X | |
| 8 | | Problems encountered |

Table 6.1: Implementation Results

handler in an inner block. Also, if an inner block has a `COMPLETE BEFORE` deadline which has been reset due to an earlier `COMPLETE BEFORE` deadline of an outer block, verify that the handler for the outer block is invoked even if the missed deadline is detected in the inner block.

**Test 8 - Timing Constraint Violations in Condition Handlers** This test verifies that even if a timing constraint is violated during the execution of a constraint handler, that control is properly passed to the handler for the timing constraint violation.

### 6.1.2 Summary of Test Results

Table 6.1.1 summarizes the results of the test suite described in the previous section. Many of the tests had a few variations which were tested. For example, **Test 5** examined condition handlers in nested blocks. This required testing both temporal and logical constraint violations on data in nested blocks. Also, tests were performed using a nesting depth greater than two to check that handlers in more distant outer blocks would also work.

In **Test 4**, we wanted to create a scenario where only some of the data processed caused a temporal constraint violation. To accomplish this, data was entered into the table at regular intervals by placing an INSERT statement with a START AFTER clause in a loop. For example:

```
my_handler() {
  printf("This is the local handler for stock.price_avi\n");
}

main() {
  int i;
  float k;

  for (k=1; k<=10; k++) {
    j = 0.5 * k;
    EXEC SQL BEGIN
      INSERT INTO stock (id, price) VALUES (:k, :j)
      START AFTER CURRENT_TIMESTAMP + INTERVAL '1' SECOND;
    END;
  }
  EXEC SQL BEGIN
    DECLARE CONTINUE HANDLER FOR stock.price_avi my_handler();
    SELECT * FROM stock;
  END;
}
```

The INSERT statement creates a timestamp value for an item if it is required. In this case, the price attribute was specified to have a timestamp. Each time the INSERT statement executes, it waits one second before it does the insertion. So as each row of the table is inserted, the timestamp of the price attribute will be approximately one second older than the timestamp of the price on the previous row.

The price attribute has a temporal consistency constraint price_avi specified that states that its value should be no more than five seconds old. When the SELECT statement is executed, the first six rows have price values that violate the price_avi constraint. Figure 6.1 shows the timeline corresponding to the situation.

To test the timing constraints on actions required slowing down portions of the code built by the preprocessor by using a routine called busy_wait(n) that executes repeated floating point divisions. An integer value $n$ is supplied to the routine, and used to generate a delay of approximately $n$ seconds. For example, suppose we have the following statement:

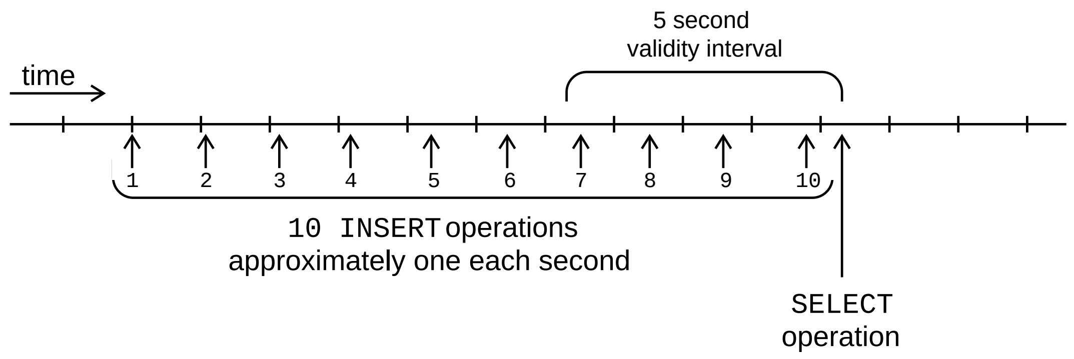
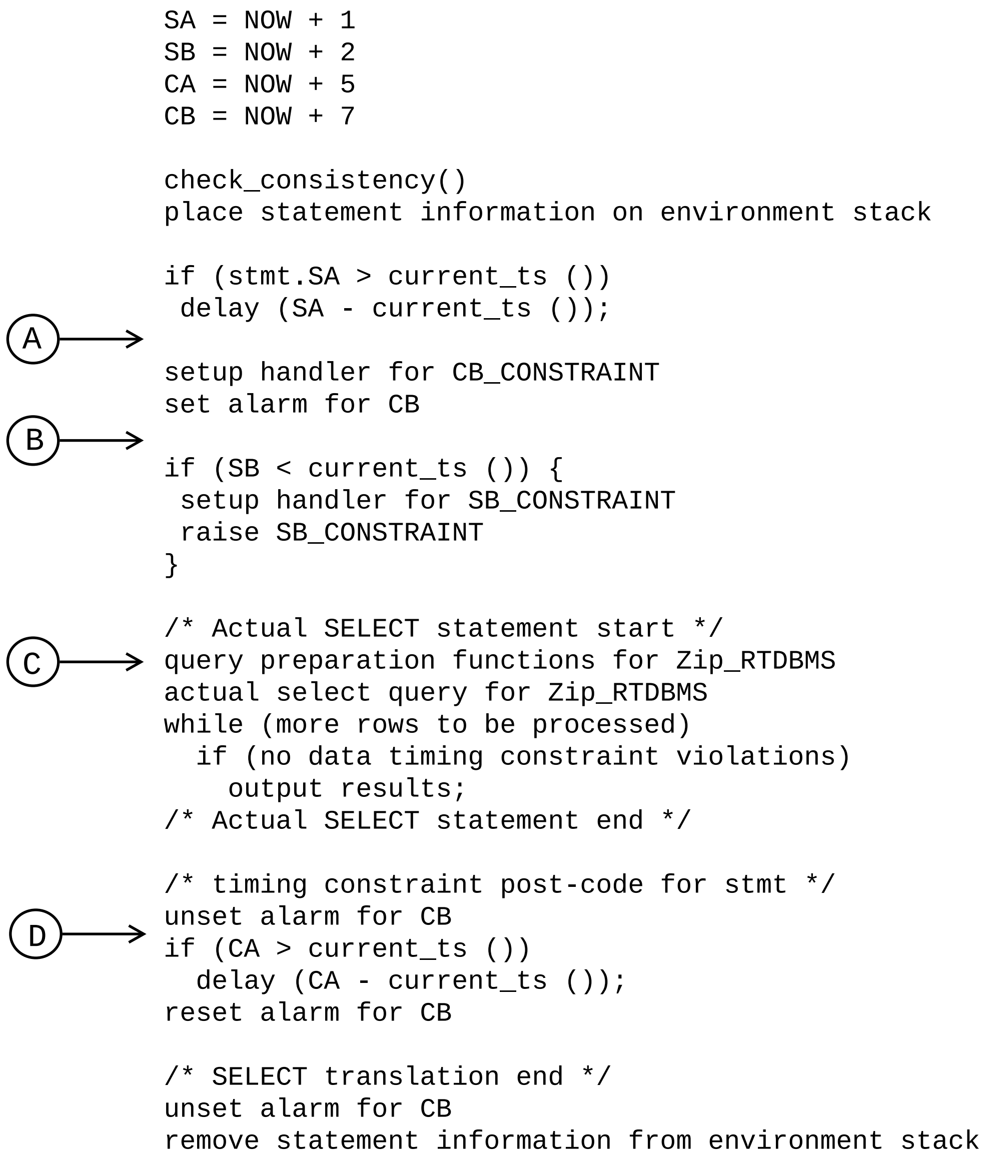


Figure 6.1: Timeline of Events for Test 4

```
SELECT id FROM stock
  START AFTER CURRENT_TIMESTAMP + INTERVAL '1' SECOND
  START BEFORE CURRENT_TIMESTAMP + INTERVAL '2' SECOND
  COMPLETE AFTER CURRENT_TIMESTAMP + INTERVAL '5' SECOND
  COMPLETE BEFORE CURRENT_TIMESTAMP + INTERVAL '7' SECOND;
```

Figure 6.2 shows the psuedo code result of the above `SELECT` statement. Possible points of insertion for the `busy_wait` routine are shown. For example, at point `A`, a delay of three seconds would be sufficient to cause the `START BEFORE` constraint to be violated. At points `B` or `C`, a delay of six or more seconds would be sufficient to cause cause the `COMPLETE BEFORE` clause to be violated (note that one second is utilized by the `START AFTER` clause). A delay at point `D` demonstrates the importance of resetting the alarm after the code for the `COMPLETE AFTER` clause.

There were problems with **Test 8**, which tests for timing constraint violations during execution of condition handlers. Delays were placed in the exception handlers to force the violation of the timing constraints. If the handler was for a data constraint violation (temporal or logical), no problems were encountered. Also, in the case where the handler was for timing constraint conflicts or violation of the `START BEFORE` constraint, the tests performed as expected. But in the case where the handler was for a `COMPLETE BEFORE` clause of an inner block, a problem was detected. Recall that the `COMPLETE BEFORE` clause is translated into code which sets a handler and a timer. When the timer expires, the appropriate handler is invoked. This handler does not currently reset the timer for the outer block. Since this timer



```
      SA = NOW + 1
      SB = NOW + 2
      CA = NOW + 5
      CB = NOW + 7

      check_consistency()
      place statement information on environment stack

      if (stmt.SA > current_ts ())
       delay (SA - current_ts ());
A --> 
      setup handler for CB_CONSTRAINT
      set alarm for CB
B --> 
      if (SB < current_ts ()) {
       setup handler for SB_CONSTRAINT
       raise SB_CONSTRAINT
      }

      /* Actual SELECT statement start */
C --> query preparation functions for Zip_RTDBMS
      actual select query for Zip_RTDBMS
      while (more rows to be processed)
        if (no data timing constraint violations)
          output results;
      /* Actual SELECT statement end */

      /* timing constraint post-code for stmt */
D --> unset alarm for CB
      if (CA > current_ts ())
        delay (CA - current_ts ());
      reset alarm for CB

      /* SELECT translation end */
      unset alarm for CB
      remove statement information from environment stack
```

Figure 6.2: Examples of Insertion Points for `busy_wait(n)` Routine

| Requirement | Description | Status |
|---|---|---|
| 3.5.2.1 | Modes of real-time | PA |
| 3.5.2.2 | Real-time transactions | A |
| 3.5.2.3 | Concurrency control | NA |
| 3.5.2.4 | Temporal consistency | A |
| 3.5.2.5 | Real-time scheduling | PA |
| 3.5.2.6 | Bounded logical imprecision | NA |
| 3.5.2.7 | Bounded temporal imprecision | NA |
| 3.5.2.8 | Main memory data | A |
| 3.5.2.9 | Time fault tolerance | A |
| 3.5.2.10 | Resource utilization limits | A |
| 3.5.2.11 | Compilable DML | A |
| 3.6.2.1 | Collection of fault information | A |
| 3.6.2.2 | Retrieval of fault information | A |
| 3.6.2.6 | Fault detection thresholds | NA |
| 3.6.2.7 | Specification of fault responses | A |

(A=Addressed PA=Partially Addressed NA=Not Addressed)

Table 6.2: DISWG Requirements RTSQL Summary

is never reset, the outer block's timing constraint will not be violated as it should. This problem could be easily remedied by designing the `COMPLETE BEFORE` handler to reset the alarm before the handler routine is actually executed.

## 6.2 DISWG Requirements Evaluation

In the following paragraphs, RTSQL will be evaluated against the DISWG requirements in the areas of real-time processing and fault tolerance as described in chapter 2. Table 6.2 summarizes these results.

*3.5.2.1* ***Modes of real-time.*** The current version of RTSQL is intended for use in soft real-time systems. Some of the features of RTSQL also support the requirements of hard real-time systems. These include directives that can be used to specify resource utilization limits such as worst case execution time and bounded table sizes. With this type of information, statements and transactions could be designed to meet their timing constraints.

*3.5.2.2* ***Real-time transactions.*** RTSQL provides a more complete transaction specification than SQL. This specification provides some mechanisms for controlled relaxation of the ACID properties. For example, the RECOVERY clause allows a transaction to affect its durability in the system by specifying a recovery level. Preconditions and postconditions can be used to relax atomicity. They can be used to specify that a transaction may commit even if some of it's subtransactions have not been able to commit (allowing for partial execution of a transaction). Transactions no longer necessarily run in isolation from one another, one transaction may be dependent upon another committing before it commits.

Timing constraints can be specified on actions (statements and transactions). These timing constraints can be used to specify an interval in which an action must start execution and an interval in which an action must complete execution. Criticality of an action can be specified through a directive.

*3.5.2.3* ***Concurrency control correctness criteria.*** The current version of RT-SQL does not address this requirement directly. SQL provides some primitive mechanisms for specifying concurrency control correctness criteria other than serializability. For example, it is possible to allow transaction one to read data written by transaction two even though transaction two has not yet committed (this phenomenon is known as dirty reads). Future versions may attempt to incorporate efforts such as the concurrency control mechanism developed in[DiP95].

*3.5.2.4* ***Temporal consistency.*** RTSQL has extended the notion of data constraints to time. This allows the user to specify the temporal consistency requirements of data.

*3.5.2.5* ***Real-time scheduling.*** RTSQL does not specify the real-time scheduling to be used in the DBMS. In keeping with the spirit of SQL, the language should not specify 'how' something should be done, but simply 'what' needs to be done. It is left to the implementation to determine the best way to accomplish the task. Thus, RTSQL attempts to provide sufficient information (such as worst case execution time, criticality, and deadlines) that could be synthesized by a number of real-time scheduling algorithms in an attempt to maintain logical and temporal consistency of the data.

*3.5.2.6* ***Bounded logical imprecision.*** RTSQL does not address imprecise data. This issue is tightly bound with the concurrency control mechanism used by the system. Imprecision is usually introduced into data because of a tradeoff to maintain temporal consistency of the data. RTSQL could easily be extended to maintain an imprecision value in a manner similar to timestamps. And, as with the timestamp of an attribute, a function for retrieving the imprecision associated with a value could be provided and utilized in constraint specifications.

*3.5.2.7* ***Bounded temporal imprecision.*** RTSQL does not address temporal imprecision of data directly. Temporal imprecision is tightly bound with the concurrency control mechanism used by the system. Constraints are used to specify temporal consistency requirements, and condition handlers are used to specify the actions to be taken when a constraint is violated. If a data value is allowed to be temporally imprecise, a handler may simply specify no action, and allow an action to continue. Though this methodology does not quantify temporal imprecision, it recognizes that it can exist.

*3.5.2.8* ***Main memory data.*** RTSQL provides a directive for specifying that a table should be located in main memory. It even goes one step further in providing a directive which could be used to specify a particular location within memory.

*3.5.2.9* ***Time fault tolerance.*** RTSQL utilizes the condition handling mechanism of SQL/PSM (with minor enhancements) to support time fault tolerance.

*3.5.2.10* ***Resource utilization limits.*** RTSQL provides some mechanisms for specifying resource limits. For example, we can specify the worst case execution time of a statement or transaction. Also, table sizes can be bounded, which provides the system with the worst case storage requirements. Bounded tables also allow computation of worst case execution times when coupled with predictable storage access, as is the case when data is stored in main memory.

*3.5.2.11* ***Compilable DML.*** RTSQL does not directly address the issue of compilable DML. This capability exists for SQL/PSM, which, given the support SQL/PSM provides for condition handling, is a likely candidate for RTSQL to actually be based upon.

*3.6.2.1* ***Collection of fault information.*** SQL provides a diagnostics area for storing information related to faults in the database system. When a statement completes execution, information related to the completion status of the statement is stored in this area. The diagnostics area is capable of recording information for more than one fault should that situation occur. Given that RTSQL is based upon SQL, it too will have this capability.

*3.6.2.2* ***Retrieval of fault information.*** SQl provides a `GET DIAGNOSTICS` statement for retrieving information from the diagnostics area. This would also be available in RTSQL.

*3.6.2.6* ***Fault detection thresholds.*** This requirement was not addressed by RTSQL.

*3.6.2.7* ***Specification of fault responses.*** This requirement is satisfied by the condition handling mechanism described in this work. The user can specify handlers that should be executed when a particular fault occurs.

**Summary of DISWG Results.** The majority of the DISWG requirements have been satisfied by RTSQL. The major weakness of RTSQL is in the area of imprecise values. The RTSORAC model provides a basis for logical imprecision to be associated with values, and the constraint mechanism can be used to bound imprecision. RTSQL constructs could be developed from this basis to support logical imprecision. Temporal imprecision must be added to both the RTSORAC model and RTSQL. SQL and RTSQL are also weak in providing mechanisms for specifying different concurrency control correctness criteria.

## 6.3 Standards Work

Preliminary results of the RTSQL efforts were presented to the DISWG committee in the fall of 1994. In general, the work was well received. Their plan was to use the proposed RTSQL to develop a standard that could be used by the Navy. The initial phase of this project was to determine existing standards which were related to the requirements document and evaluating them in the context of the requirements. Appendix C shows the results of this study. Two standards were examined in detail: Remote Data Access (RDA) and SQL2. The table in Appendix C notes if the requirement is addressed, and if so, where the related material appears in the standard that addresses the requirement.

Once this phase was completed, requirements that were not addressed or only partially addressed were divided into different categories. Subgroups were formed to examine each category in detail. Just as this effort was starting, it was placed on hold due to budgetary considerations. DISWG's funding was substantially curtailed to be reevaluated in the next fiscal year.

The long term goal of the DISWG committee was to have RTSQL recognized as an international standard. To attain this goal, the committee had representatives become members of the ANSI X3 committee. We were allowed to participate in this process. When the DISWG funding was reduced, this effort was continued, since establishing a standard is understood to be a long, complicated process.

We have become fully involved in the ANSI X3 committee as they work on the next standard SQL3. We have provided input to issues which would eventually impact RTSQL, such as condition handling. We have also had the opportunity to request that a working group be formed under the auspices of ANSI to study RTSQL. This involved a formal request, and a technical presentation of our preliminary results in May of 1995. A number of members representing a diverse customer base have encouraged us in our efforts. These include representatives for the petroleum industry and financial services industry.

At this point in time, we have been asked by the standards committee to generate a base document for RTSQL. The intent is to have this document presented at the international level to see the level of interest. Though ANSI has the power to form a subgroup to study RTSQL, they prefer to keep in line with the interests of the international community. If a sufficient number of countries support the idea, the initial work will be under the auspices of ANSI for later acceptance by ISO (International Standards Organization).

# Chapter 7

# Conclusion

Our goal for this work was to develop a set of language constructs which could be used as a basis for creating a standard query language for real-time database systems. The contributions we have made in reaching this goal include the definition of the RTSORAC model, the specification of RTSQL, and the demonstration of completeness and feasibility of the resulting language. These contributions have provided a strong foundation for the development of a standards document that could be submitted to the standards committee for review. In this chapter we summarize these contributions, and discuss their limitations and possible future efforts.

## 7.1 Contributions

Recall that the development of RTSQL included specification of language constructs, an evaluation of their feasibility through implementation, and evaluation of the specification for completeness. Each of these areas is summarized in the following paragraphs.

**Language Specification.** The language constructs of RTSQL fall into three categories. The first category is constraints. Constraints are used to specify the semantics of correctness (with respect to time) of data, operations, and transactions. Absolute and relative temporal consistency are supported by an extension to SQL2's data definition language allowing the specification of timestamps on data. An amendment to SQL2's constraint mechanism allows use of datetime functions, which, along with the timestamps, are sufficient to specify temporal consistency constraints. Timing constraints on general data manipulation statements, including transactions, are specified with a complete set of timing constraints including: start times, deadlines and periods. The exception handling mechanism was also extended to support specification of handlers for timing constraints and user-defined data constraints.

The second category is directives. Directives are used to specify assertions about data, operations, and transactions. Database partitioning, which can allow higher data availability, is supported by the addition of a `DEPENDS ON` data definition directive. Relative importance of transactions is a replaceable characteristic specified by a `IMPORTANCE LEVEL` directive in a transaction definition. Directives are used to specify system-dependent configurations that can affect predictability of execution, such as whether a data item is kept in main memory only.

The third category is transactions. The transaction specification provides mechanisms for controlled relaxation of the ACID properties and more flexible transaction structure. These capabilities allow specification of various transaction and subtransaction structure that can facilitate early commitment and therefore increase data availability and flexibility in real-time scheduling.

**Language Completeness.** RTSQL was evaluated in the context of the DISWG requirements. Recall that the DISWG requirements were quite extensive, and that RTSQL addresses only two of the areas in depth. Many of the other requirements are covered by the SQL standard and the RDA standards. The evaluation of RTSQL was focused on two areas of the requirements document: real-time processing and fault tolerance. In the area of real-time processing, most of the requirements were addressed by RTSQL including support for temporal data, real-time transactions, time fault tolerance, and resource utilization limits. A subset of the fault tolerance requirements were addressed including collection and retrieval of fault information, fault detection thresholds, and specification of fault responses through the use of condition handling.

**Language Feasibility.** An implementation was done to determine the feasibility of some of the proposed features. The focus of the implementation was on data timing constraints, timing constraints on actions, and condition handlers for these constraints if they are violated. The testing of these constructs produced favorable results. Users could define time constrained data and condition handlers which were invoked if these constraints were violated. They could also place timing constraints on statements and blocks and define handlers which were invoked if these constraints were violated. Provisions were made to handle conflicting timing constraints as well.

## 7.2 Limitations and Future Work

The work described in this dissertation provides a strong foundation for a standard query language for real-time databases. However, there are some limitations to the work upon which we can focus future efforts. The following paragraphs will highlight some of the known limitations of the RTSORAC model and RTSQL. Also, issues for extending the implementation will be discussed.

### 7.2.1 RTSORAC Model.

The RTSORAC model has no support for temporal imprecision. Related research [DiP95] has focused on concurrency control and bounding of logical imprecision, little has been done in the study of temporal imprecision. Given that the RTSORAC model provides the compatibility function which is used in concurrency control, there is a basis that could be used to study this issue.

Future efforts could also include formalization of the RTSORAC model. As mentioned in Chapter 3, RTSORAC is loosely based upon the entity relationship model. Extensive work has been done to formalize the ER model[NP88]. There have also been efforts to add time to the relational model in the work on temporal databases [MS87]. In the context of these related efforts, one area of future work would be to formalize the RTSORAC model, including the aspect of time.

### 7.2.2 RTSQL.

RTSQL does not provide any mechanism for handling historical data. Time constrained data is often the result of information being gathered at regular intervals from a sensor or other input device. In this case, there is not only the most recent value from the sensor that is of interest, but some historical record of values that should be maintained. For example, these values may be needed to forecast future values through extrapolation or to evaluate past performance. Though the maintenance of historical data is usually a feature of temporal databases, it seems that RTSQL should provide mechanisms for handling this situation.

The recovery mechanism provided for RTSQL transactions must be improved. In the current version of the language, the user can specify three levels of recovery for a transaction: automatic, user-specified (where an alternative transaction is specified), or manual. Recovery needs to be enhanced because of the introduction of full and flexible transaction capabilities, and because of timing and predictability considerations. In particular, mechanisms that allow for forward recovery instead of just traditional database backward recovery should be introduced. Also, the semantics of recovery for subtransactions must be explored.

One of the DISWG requirements that was not addressed directly by RTSQL was concurrency control correctness criteria. SQL does provide very primitive mechanisms that can be specified on the transaction level which allow more concurrent access to data while sacrificing consistency. The RTSORAC model and work done in [DiP95] provide a strong foundation on which extensions could be based using the object paradigm, but further research would be necessary using the relational model.

Directives could be expanded to include worst case resource requirements for memory needed by to execute the transaction.

### 7.2.3 Implementation.

The implementation presented here was a subset of the proposed constructs. Some of these, such as transactions, will require extensive research and work. Others, such as some of the directives, will require little effort to support.

The implementation provides a good starting point in exploring the implications of maintaining time constrained data using the active approach as discussed in Section 4.1.1. This is related to the work in the area of active databases[MD89]. In an active database, one action may cause another action to be initiated, and so forth. In real-time databases the passage of time as well as other actions can cause another action to be initiated. The active approach of maintaining time constrained data should be fully explored.

The work on RTSQL presented here represents the initial efforts in establishing a standard query language for real-time databases. With respect to creating an actual standard, RTSQL is in its infancy. Much work must be done to get the work in a form acceptable to the standards community. This includes identification of specific sections of the SQL2 standards document which are affected by the proposed standards. There are also specific rules on how to structure the document in terms of necessary sections and their contents. But the contributions of this work: the RT-SORAC model, the specification of RTSQL, and the demonstration of completeness and feasibility of the specification provide a strong foundation for the establishment of a standard query language for real-time databases.

# References

[BHG87] P. Bernstein, V. Hadzilacas, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Massachusetts, 1987.

[BMHD89] A.P. Buchmann, D.R. McCarthy, M. Hsu, and U. Dayal. Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control. In *The Fifth International Conference on Data Engineering*, February 1989.

[Boo91] Grady Booch. *Object-Oriented Design.* The Benjamin/Cummings Publishing Company, Redwood City, CA, 1991.

[Che76] P.P. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, 1(1), March 1976.

[DD92] C. Date and H. Darwen. *A Guide to SQL Standard.* Addison-Wesley Publishing, Reading, MA., 1992.

[DG91] Oscar Diaz and Peter M. D. Gray. Semantic-rich user-defined relationship as a main constructor in object-oriented databases. In R.A. Meersman, W. Dent, and S. Khosla, editors, *Object-Oriented Databases: Analysis,Design & Construction (DS4)*, pages 207 – 224. Elsevier Science Publishers, B.V. (North-Holland), 1991.

[DiP95] Lisa Cingier DiPippo. *Object-based semantic real-time concurrency control.* PhD thesis, University of Rhode Island, 1995.

[DP93] Pamela Drew and Calton Pu. Asynchronous consistency restoration under epsilon serializability. Technical Report OGI-CSE-93-004, Department of Computer Science and Engineering, Oregon Graduate Institute, 1993.

[DW93] L. Cingiser DiPippo and V. Fay Wolfe. Object-based semantic real-time concurrency control. *Proceedings of the 14th IEEE Real-time Systems Symposium*, December 1993.

[For93] P. Fortier. *Early Commit.* PhD thesis, University of Massachusetts Lowell, 1993.

[For94] P. Fortier. ANSI DBSSG PRISTG: Real-time database management systems reference model. *ANSI DBSSG Predictable Real-time Information Systems Task Group, PRISTG Document No. 94-001*, January 1994.

[FS94] P. Fortier and Cdr. G. Sawyer. DISWG a new player in NGCR open systems standards. *To appear in Computer Standards and Interfaces*, 1994.

[FWP94] Paul Fortier, Victor Fay Wolfe, and JJ Prichard. Flexible real-time SQL transactions. In *IEEE Real-Time Systems Symposium*, Dec. 1994.

[Gal91] L. Gallagher. Database management standards: Status and applicability. *Computer Standards and Interfaces*, 12, 1991.

[Gal92] L. Gallagher. Object SQL: Language extentions for object data management. In *International Society for Mini and Microcomputers Conference on Information and Knowledge Management*, August 1992.

[GMGK+91] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Modeling long-running activities as nested sagas. *Bulletin of the IEEE Technical Committee on Data Engineering*, 14(1), March 1991.

[Gor93] K. Gordon. *DISWG Database Management Systems Requirements.* NGCR SPAWAR 331 2B2, Alexandria, Virginia, 1993.

[Joh75] S. C. Johnson. YACC-yet another compiler compiler. Technical Report CSTR 32, Bell Laboratories, Murray Hill, N. J., 1975.

[KS86] Eugene Kligerman and Alexander Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, September 1986.

[Les75] M. E. Lesk. LEX-a lexical analyzer generator. Technical Report CSTR 39, Bell Laboratories, Murray Hill, N. J., 1975.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LMB95] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, 1995.

[LN88] Kwei-Jay Lin and Swaminathan Natarajan. Expressing and maintaining timing constraints in FLEX. In *IEEE Real-Time Systems Symposium*, pages 96–105, December 1988.

[MD89] Dennis R. McCarthy and Umeshwar Dayal. The architecture of an active database management system. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Portland Oregon, June 1989.

[Mel92] J. Melton, editor. *ANSI X3.135-1992, American national Standard, Database Language SQL*. American National Standards Institute, 1992.

[Mel95] J. Melton, editor. *ISO/IEC JTC1/SC21/WG3 DBL YOW-006 and ANSI X3H2-95-086, (ISO/ANSI working draft) SQL Persistent Stored Modules (SQL/PSM)*. American National Standards Institute, March 1995.

[MS87] Edwin McKenzie and Richard Snodgrass. Extending the relational algebra to support transaction time. In *Proceedings of the 1987 ACM SIGMOD International Conference on the Management of Data*, San Francisco, California, May 1987.

[MS92] J. Melton and A. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kauffman Publishers, San Mateo, CA., 1992.

[NP88] P.A. Ng and J. F. Paul. *A formal definition of entity-relationship models*. North Holland, Amsterdam, 1988.

[OV91] T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall Inc., Englewood Cliffs, New Jersey, 1991.

[PDPW94] JJ Prichard, Lisa Cingiser DiPippo, Joan Peckham, and Victor Fay Wolfe. RTSORAC: A real-time object-oriented database model. In *Proceedings of the International Conference on Database and Expert Systems Applications*, September 1994.

[PM94] W. Pugh and T. Marlow, editors. *Proceedings of the ACM SIGPLAN workshop on language, compiler and tool support for real-time systems.* Association for Computing Machinery, June 1994.

[Ram93] Krithi Ramamritham. Real-time databases. *International Journal of Distributed and Paralled Databases*, 1(2), 1993.

[RDA] *ISO/IEC 9579-2, Information Technology, Remote Database Access, Part 2: SQL Specialization.* American National Standards Institute.

[RP95] Krithi Ramamritham and Calton Pu. A formal characterization of epsilon serializability. To appear in *IEEE Transactions on Knowledge and Data Engineering.* Also available as technical report No. CUCS-044-91 at Department of Computer Science, Columbia University, 1995.

[Sha85] L. Sha. *Modular Concurrency Control and Failure Recovery – Consistency, Correctness and Optimality.* PhD thesis, Carnegie-Mellon University, 1985.

[Sno94] R. Snodgrass et. al. TSQL2 language specification. *ACM SIGMOD Record*, 23(1):65–86, March 1994.

[Son90] Sang H. Son. Real-time database systems: A new challenge. *Data Engineering*, 13(4), December 1990.

[ST90] P. Stachour and B. Thurasingham. SQL extensions for security assertions. *Computer Standards & Interfaces*, 11(1), 1990.

[Ulu92] Ozgur Ulusoy. Current research on real-time databases. *SIGMOD Record*, 21(4):16 – 21, December 1992.

[WDL93] Victor Wolfe, Susan Davidson, and Insup Lee. *RTC*: Language support for real-time concurrency. *Real-Time Systems*, 5(1):63–87, March 1993.

[YWLS94] P. Yu, K. Wu, K. Lin, and S.Son. On real-time databases: Concurrency control and scheduling. *Proceedings of the IEEE*, 82, January 1994.

# Appendix A

# RTSQL grammar

This is a copy of the actual grammar used in the implementation.

```
sql_list:
                sql_statement ';'
        |       sql_list sql_statement ';'
        ;

sql_statement:
                base_table_def
        |       select_statement
        |       insert_statement
        |       update_statement
        |       delete_statement
        |       compound_statement
        ;

/*
** RTSQL compound statement
*/

compound_statement:
                BEGIN
                opt_handler_list
                sql_statement_list
                END
                opt_timing_constraint
        ;
```

```
sql_statement_list:
                sql_statement ';'
        |       sql_statement_list sql_statement ';'
        ;

opt_handler_list:
                /* empty */
        |       handler_list

handler_list:
                handler ';'
        |       handler_list handler ';'
        ;

handler:
                DECLARE handler_type HANDLER
                FOR full_name handler_action
        ;

handler_type:
                CONTINUE
        |       EXIT
        ;

handler_action:
                routine_name '(' ')'
        ;

/*
**  RTSQL CREATE TABLE statement
*/

base_table_def:
                CREATE TABLE table_name
                '(' base_table_element_commalist ')'
        ;

base_table_element_commalist:
                base_table_element
        |       base_table_element_commalist ',' base_table_element
        ;

base_table_element:
                column_def
        |       table_constraint_def
        ;

column_def:
                column_name data_type opt_timestamp
        ;

opt_timestamp:
                /*empty*/
        |       WITH TIMESTAMP
        ;

table_constraint_def:
                CONSTRAINT constraint_name column_constraint
        ;

column_constraint:
                CHECK '(' constraint_search_condition ')'
        ;


/*
**  RTSQL SELECT statement
*/

select_statement:
                SELECT select_list table_exp
                opt_timing_constraint
        ;

select_list:
                '*'
        |       column_commalist
        ;
```

```
table_exp:
                from_clause
                opt_where_clause
        ;

from_clause:
                FROM table_name
        ;

opt_where_clause:
                /* empty */
        |       where_clause
        ;

where_clause:
                WHERE search_condition
        ;


/*
**  RTSQL INSERT statement
*/

insert_statement:
                INSERT INTO table_name opt_column_commalist
                values_or_query_spec
                opt_timing_constraint
        ;

opt_column_commalist:
                /* empty */
        |       '(' column_commalist ')'
        ;

column_commalist:
                column_name
        |       column_commalist ',' column_name
        ;

values_or_query_spec:
```

```
                VALUES '(' insert_atom_commalist ')'
        ;

insert_atom_commalist:
                insert_atom
        |       insert_atom_commalist ',' insert_atom
        ;
insert_atom:
                atom
        ;

atom:
                literal
        |       parameter
        ;


/*
**  RTSQL UPDATE statement
*/
update_statement:
                UPDATE table_name SET assignment_commalist
                opt_where_clause
                opt_timing_constraint
        ;

assignment_commalist:
                assignment
        |       assignment_commalist ',' assignment
        ;

assignment:
                column_name '=' numeric_exp
        ;

/*
**  RTSQL DELETE statement
*/
delete_statement:
                DELETE FROM table_name opt_where_clause
                opt_timing_constraint
        ;

/*
** RTSQL search_condition clause
*/

search_condition:
                search_condition OR search_condition
        |       search_condition AND search_condition
        |       NOT search_condition
        |       '(' search_condition ')'
        |       predicate
        ;

predicate:
                comparison_predicate
        ;

comparison_predicate:
                numeric_exp COMPARISON numeric_exp
        |       datetime_exp COMPARISON datetime_exp
        ;

numeric_exp:
                numeric_exp '+' numeric_exp
        |       numeric_exp '-' numeric_exp
        |       numeric_exp '*' numeric_exp
        |       numeric_exp '/' numeric_exp
        |       '+' numeric_exp %prec UMINUS
        |       '-' numeric_exp %prec UMINUS
        |       '(' numeric_exp ')'
        |       numeric_primary
        ;

numeric_primary:
                INTNUM
        |       APPROXNUM
        |       column_name
        ;
```

```
datetime_exp:
                interval_exp '+' datetime_exp
        |       datetime_exp '+' interval_exp
        |       datetime_exp '-' interval_exp
        |       datetime_primary
        ;

interval_exp:
                interval_literal
        |       '(' datetime_exp '-' datetime_exp ')' DAY TO SECOND
        ;

interval_literal:
                INTERVAL STRING interval_qualifier
        ;

interval_qualifier:
                DAY TO SECOND
        |       SECOND
        ;

datetime_primary:
                datetime_literal
        |       datetime_function_ref
        |       TIMESTAMP '(' column_name ')'
        ;


datetime_literal:
                TIME STRING
        |       TIMESTAMP STRING
        ;


/*
** RTSQL search_condition clause for constraints
*/

constraint_search_condition:
```

```
                constraint_search_condition OR
                  constraint_search_condition
        |       constraint_search_condition AND
                  constraint_search_condition
        |       NOT constraint_search_condition
        |       '(' constraint_search_condition ')'
        |       constraint_predicate
        ;

constraint_predicate:
                constraint_comparison_predicate
        ;

constraint_comparison_predicate:
                numeric_exp COMPARISON numeric_exp
        |       datetime_exp COMPARISON datetime_exp
        ;


/*
** RTSQL opt_timing_constraint clause
*/

opt_timing_constraint:
                /* empty */
        |       timing_constraint_list
        ;

timing_constraint_list:
                timing_constraint
        |       timing_constraint_list timing_constraint
        ;

timing_constraint:
                START AFTER datetime_exp
        |       START BEFORE datetime_exp
        |       COMPLETE AFTER datetime_exp
        |       COMPLETE BEFORE datetime_exp
        |       PERIOD interval_exp opt_start_at opt_until
        ;
```

```
opt_start_at:
                /* empty */
        |       START AT datetime_exp
        ;

opt_until:
                /* empty */
        |       UNTIL search_condition
        ;

/*
** RTSQL miscellaneous
*/

table_name:
                NAME
        ;

column_name:
                NAME
        ;

constraint_name:
                NAME
        ;

routine_name:
                NAME
        ;

full_name:
                NAME '.' NAME
        ;

literal:
                INTNUM
        |       APPROXNUM
        ;
```

```
parameter:
                PARAMETER
        ;

datetime_function_ref:
                CURRENT_TIMESTAMP
        ;

                /* data types */

data_type:
                INTEGER
        |       REAL
        |       CHARACTER '(' INTNUM ')' }
        ;
```

# Appendix B

# Using the RTSQL Preprocessor

This guide will explain how to use the RTSQL preprocessor with the Zip_RTDBMS database system.

## B.1 Using Zip_RTDBMS

### B.1.1 Starting the Zip Database Server `zerver`

In order to use the Zip_RTDBMS database server you must start the Zip_RTDBMS server process `zerver`. The command for starting this process is in the bin directory of the home directory for Zip_RTDBMS, referred to here as `$(zipHome)`. Change directory to the `bin` directory in `$(zipHome)` as follows:

```
cd $(zipHome)/bin
```

This is very important, the Zip_RTDBMS database server will not run correctly if it is not started from this directory. To start the server as a background process, enter the command:

```
zerver &
```

There is no advantage to running it in the foreground.

If the Zip_RTDBMS database server zerver was not shutdown properly previously, you will see a message similar to the following:

```
zerver: master server message queue already exists.
>>File already exists
The master server process may already be running.
>>>Normal termination
```

You will need to remove one or more files from the `/tmp` directory. Use the following command:

```
rm -f /tmp/.Zerver_MQ /tmp/.MQ*
```

If you are unable to delete these files (they are owned by another user), ask the system administrator to do it for you. Once these files have been deleted, the server zerver may be started as shown above.

### B.1.2 Creating a Zip_RTDBMS Schema File

In Zip_RTDBMS, a schema file is created by using the command `Zip_parser`. This command takes an ascii file with Zip_RTDBMS data definition commands and creates a binary file used by the `zerver` process. As with starting the `zerver` process, you must first change directories to `$(zipHome)/parser`. For example, if you have a Zip_RTDBMS data definition file in your home directory called `tst.DDL` then you could enter:

```
cd $(zipHome)/parser
Zip_parser < ~/tst.DDL
```

If the database name in the file is noted as *stocks* in the `/tmp` directory, then this will create the schema file `/tmp/stocks.schema`.

Note that this simply creates the schema file, an database instance must still be created. This can be done in two ways. The first is from a program using the appropriate Zip_RTDBMS library calls. The second is to use the Zip_RTDBMS `create_database` command. For example, to create an empty database from the schema file `/tmp/stocks.schema`:

```
$(zipHome)/bin/create_database /tmp/stocks.schema
```

This will create a database file `/tmp/stock`.

## B.2 Using the RTSQL Preprocessor

The RTSQL preprocessor will translate a C++ program containing embedded RT-SQL statements to a C++ program with library calls to the Zip_RTDBMS interface. The resulting program must then be linked with the Zip_RTDBMS interface library and with the Zip_RTDBMS libraries.

Two files must be created by the user. The first is a file containing all of the `CREATE TABLE` statements. It is assumed that this file will have the suffix `.EDD` (extended data definition). For example:

```
/*
** File tst.EDD
*/
/* stock table */
EXEC SQL CREATE TABLE stock (
id INTEGER,
price   REAL WITH TIMESTAMP,
CONSTRAINT check_id CHECK ((id > 0 AND id <= 99) AND price > 0.0),
CONSTRAINT price_ok CHECK (price <= 1000.0),
CONSTRAINT check_price_avi
    CHECK (TIMESTAMP(price) < CURRENT_TIMESTAMP + INTERVAL '5' SECOND)
);
```

The second file is a C++ program file that contains embedded data manipulation statements. It is assumed that this file will have the suffix `.EDM` (extended data manipulation). For example:

```
/*
** File tst.EDM
*/
#include "tstEDD.h"
main() {
  int i;
  float j;
  for (i=1; i<=100; i++) {
     j = i / 2.0;
     EXEC SQL INSERT INTO stock ( id, price ) VALUES ( :i, :j );
  }
  EXEC SQL SELECT * FROM stock;
  printf("That's all folks...\n");
}
```

Note the the `tst.EDM` file contains the include statement `#include "tstEDD.h"`. The filename for the `#include` must be of the form `xxxEDD.h`, where `xxx.EDD` is the corresponding data definition file.

To use the RTSQL preprocessor, use the `rtsql` command on the data manipulation file. For example:

```
rtsql tst.EDM
```

This will create the following files:

**tst.DDL** This file contains Zip_RTDBMS data definition commands. It must be run through the `Zip_parser` command to create a schema file for Zip_RTDBMS.

**tstEDM.cc** This file contains the translation of the tst.EDM file from embedded RTSQL to C++ code with library calls to the Zip_RTDBMS interface.

**tstEDD.h** This file contains the class definition corresponding to the CREATE TABLE command in the tst.EDD file. These classes are derived from the base classes for tables and attributes that are contained in the Zip_RTDBMS interface. Recall that it was this filename that appeared in an `#include` statement in **tstEDM.cc**.

**tstEXC.h** This file contains classes corresponding to the RTSQL blocks specified in **tst.EDM**. Each class stores the condition handlers and timing constraints associated with a block. This file is is in an `#include` statement of **tstEDD.h**.

To compile the resulting code, you must link with the Zip_RTDBMS interface library **ZipInterfaceLib.a** and the Zip_RTDBMS libraries **ZipLib.a** and **ZipCommonLib.a** as follows:

```
g++ -X tstEDM.cc ZipInterfaceLib.a ZipLib.a ZipCommonLib.a
```

The `-X` flag is particular to the LynxOS.

# Appendix C

# DISWG Requirements

This is a summary of a study done by the NGCR DISWG committee to evaluate their requirements against two existing standards: Remote Data Access (RDA) and SQL2. The table notes if the requirement is addressed, and if so, where the related material appears in the standard that addresses the requirement.

| SECTION | RDA FUNCTION(S) | SQL2 FUNCTION(S) |
|---|---|---|
| General Requirements (3.1) | | |
| 3.1.2.1 Public Specifications | Yes | Yes |
| 3.1.2.2 Portability | Yes | Introduction, Section 4.33(Leveling), Section 4.34 (SQL Flagger) |
| 3.1.2.3 Interoperability | Yes | Section 4.34 (SQL Flagger) |
| 3.1.2.4 Supportability | Yes | Yes |
| 3.1.2.5 Hardware Independent | Yes | Yes |
| 3.1.2.6 OS Independent | Yes | Yes |
| 3.1.2.7 Network Independent | Yes | Yes |
| 3.1.2.8 Programming Language Independent | Yes | Section 4.23 (Embedded Syntax),{Supports Ada, C, COBOL, FORTRAN, MUMPS, PL/1} |
| 3.1.2.9 DBMS Independent | Yes | Yes |
| 3.1.2.10 Scalability | Yes | Yes |
| 3.1.2.11 Modularity | Unclear | Yes |
| 3.1.2.12 Extensibility | Yes | Yes |
| 3.1.2.13 Uniformity | Yes | Yes |
| 3.1.2.14 Configurability | Yes | Yes |
| 3.1.2.15 Compatability with other NGCR Stds | Undetermined | Undetermined |

| SECTION | RDA FUNCTION(S) | SQL2 FUNCTION(S) |
|---|---|---|
| Basic DBMS Services (3.2) | | |
| 3.2.2.1 Persistent Data | Part 1 Section 4.1 (Server Execution) | Section 13 (Data Manipulation), Section 17 (Dynamic SQL) |
| 3.2.2.2 Multiple Users | Yes | Section 16 (Session Management) |
| 3.2.2.3 Conventional Data Types | Part 1 Section 3.1.5.1.1(Argument Spec.) Not Explicit. | Section 4.1 (Data Types) |
| 3.2.2.4 BLOBs | Part 1 Section 3.1.5.1.1 (Argument Spec.). Not Explicit. | |
| 3.2.2.5 Expressiveness of DML | Yes | Section 13 (Data Manipulation) |
| 3.2.2.6 Planned Queries | Yes | Section 13 (Data Manipulation) |
| 3.2.2.7 Ad hoc Queries | Yes | Section 13 (Data Manipulation) |
| 3.2.2.8 Interactive Queries | Yes | Section 13 (Data Manipulation) |
| 3.2.2.9 Embedded Queries | Yes | Ada, C, COBOL, FORTRAN, MUMPS, PL/1 |
| 3.2.2.10 Compiled Queries | Yes | Section 13 (Data Manipulation) |
| 3.2.2.11 Interpreted Queries | Yes | Section 13 (Data Manipulation) |
| 3.2.2.12 Transactions | Part 1 Section 3.1.2 (Trans Mgmt Svcs) & Section 3.1.3 (Control Svcs.) | Section 14 (Transaction Management) |
| 3.2.2.13 Data Models | Yes | Yes |
| 3.2.2.14 Conceptual Schema Def. | Part 1 Section 3.1.5 (Database Lang Svcs) Not Explicit | Section 11 (Schema Definition and Manipulation) |
| 3.2.2.15 External Schema Def. | Part 1 Section 3.1.5 (Database Lang Svcs) Not Explicit | Section 11 (Schema Definition and Manipulation) |
| 3.2.2.16 Internal Schema Def. | Part 1 Section 3.1.5 (Database Lang Svcs) Not Explicit | Section 11 (Schema Definition and Manipulation) |
| 3.2.2.17 Identification and Authentic | Part 1 Section 3.1.1.1.1 | Section 16 (Session Management) |
| 3.2.2.18 DAC | Part 1 Section 3.1.2.1.1 R-Open Service | Section 10.3 (Privileges) |
| 3.2.2.19 Access to metadata | Yes | Section 13 (Data Manipulation) |
| 3.2.2.20 Multiple DBMSs | Yes | Yes |
| 3.2.2.21 Multiple Databases | Yes | Section 13 (Data Manipulation) |
| 3.2.2.22 Tracing | Undetermined | Undetermined |
| 3.2.2.23 Statistical Monitoring | Part 1 Sec 4.2.2 Error Diagnostics | Section 18 (Diagnostics Management) |
| 3.2.2.24 Training Mode | Not Explicit | Not Explicit |

| SECTION | RDA FUNCTION(S) | SQL2 FUNCTION(S) |
|---|---|---|
| Distribution (3.3) | | |
| 3.3.2.1 Dist. Query Processing | | Section 15 (Connection Management), via RDA |
| 3.3.2.2 Dist. Transaction Mgmt | Part 1 Section 1.3.9 (Dist. Trans. Proc) Not Explicit | Section 15 (Connection Management), via RDA |
| 3.3.2.3 Location Transparency | Undetermined | SQL3 |
| 3.3.2.4 Fragmentation Transparency | Undetermined | |
| 3.3.2.5 Replication Transparency | Undetermined | SQL3 |
| 3.3.2.6 Data Definition | Undetermined | |
| 3.3.2.7 Local Autonomous Proc. | Yes | Yes |
| 3.3.2.8 Continuous Operation | Yes | Yes |
| 3.3.2.9 Hardware Independent | Yes | |
| 3.3.2.10 OS Independent | Yes | |
| 3.3.2.11 Network Independent | Yes | |
| Heterogeneity (3.4) | | |
| 3.4.2.1 Remote Database Access | Yes | SQL3 |
| 3.4.2.2 Global Transactions | Undetermined | |
| 3.4.2.3 Multidatabase Systems | Part 1 Section Introduction | |
| 3.4.2.4 Federated Database Systems | | |
| Real-Time Processing (3.5) | | |
| 3.5.2.1 Modes of Real-Time | | |
| 3.5.2.2 Real-Time Transactions | | |
| 3.5.2.3 Conc. Control Correctness | | |
| 3.5.2.4 Temporal Consistency | | |
| 3.5.2.5 Scheduling | | |
| 3.5.2.6 Bounded Logical Imprecision | | |
| 3.5.2.7 Bounded Temporal Imprecision | | |
| 3.5.2.8 Main Memory Data | | |
| 3.5.2.9 Time Fault Tolerance | | |
| 3.5.2.10 Resource Utilization Limits | | |
| 3.5.2.11 Compilable DBL | | |

| SECTION | RDA FUNCTION(S) | SQL2 FUNCTION(S) |
|---|---|---|
| Fault Tolerance (3.6) | | |
| 3.6.2.1 Collection of Fault Info. | | |
| 3.6.2.2 Fault Info Retrieval | | |
| 3.6.2.3 Initiate Diag Tests | Undetermined | Section 18 (Diagnostics Management) |
| 3.6.2.4 Retrieve Diag Tests | Undetermined | Section 18 (Diagnostics Management) |
| 3.6.2.5 Operational Status | Part 1 Section 4.1.2.1 Generation of RDA Operation Entities | |
| 3.6.2.6 Fault Thresholds | | |
| 3.6.2.7 Spec of Fault Responses | | |
| 3.6.2.8 Reconfiguration | Undetermined | Undetermined |
| 3.6.2.9 Replicated Components | Yes | Yes |
| Integrity (3.7) | | |
| 3.7.2.1 Domains | Part 1 Section 4.1.1 | Section 11.21 (Domain Definition) |
| 3.7.2.2 Keys | Part 1 Section 4.1.1 | Section 4.10.2 (Table Constraints) |
| 3.7.2.3 Referential Constraints | Undetermined | Undetermined |
| 3.7.2.4 Assertions | | Section 4.10.4 (Assertions) |
| 3.7.2.5 Triggers | | SQL3 |
| 3.7.2.6 Alerters | | SQL3 |
| 3.7.2.7 Manage Constraints | Part 1 pgs 64, 116, 123. Not Explicit | Section 4.10 (Integrity Constraints) |
| 3.7.2.8 Null Values | | Section 3.1 (Definitions) & Section 4.1 (Data Types) |
| Security (3.8) | | |
| 3.8.2.1 MLS | | |
| 3.8.2.2 Labeling | | SQL3 |
| 3.8.2.3 MAC | Part 1 Section 3.1.4.1.1 (Specific Access Control Data) Not Explicit | |
| 3.8.2.4 DAC | Part 1 Section 3.1.4.1.1 (Specific Access Control Data) Not Explicit | Section 10.3 (Privileges) |
| 3.8.2.5 Role Based Access Control | Part 1 Section 3.1.4.1.1 (Specific Access Control Data) Not Explicit | Section 10.3 (Privileges) |
| 3.8.2.6 Integrity | | Section 4.10 (Integrity Constraints) |
| 3.8.2.7 Consistency | Part 1 Section 5.2.2 | |
| 3.8.2.8 Identification and Authentic | Part 1 Sec 3.1.1.1.1 (User Authentic Data) | Section 16 (Session Management) |
| 3.8.2.9 Security Auditing | | SQL3 |

| SECTION | RDA FUNCTION(S) | SQL2 FUNCTION(S) |
|---|---|---|
| 3.8.2.10 Least Privilege | Part 1 Sec 4.1.1.2 | |
| 3.8.2.11 Trusted Path | | |
| 3.8.2.12 Trusted Recovery | Part 1 Section 5.1.2.2 | |
| 3.8.2.13 Inference and Aggregation | | |
| 3.8.2.14 Multilevel Data Model | | |
| 3.8.2.15 SQL Extension | | (Standard referenced in RD) |
| 3.8.2.16 OS Interface | | |
| 3.8.2.17 Network Interface | | |
| 3.8.2.18 Heterogeneity | | |
| 3.8.2.19 Next-Gen MLS | | |
| 3.8.2.20 Trusted Database Interpret | | |
| Advanced Database Management Services (3.9) | | |
| 3.9.2.1 Persistent Objects | | |
| 3.9.2.2 Object Identifiers | Part 1 Section 1.3.7 Not Explicit | SQL3 |
| 3.9.2.3 Collection Data Type | | |
| 3.9.2.4 User-Defined Data Types | Part 1 Section 3.1.5 Not Explicit | SQL3 |
| 3.9.2.5 Sorting Order | | |
| 3.9.2.6 Temporal Data | | |
| 3.9.2.7 Spatial Data | | |
| 3.9.2.8 Uncertain Data | | |
| 3.9.2.9 Derived Attributes | | |
| 3.9.2.10 Composite Objects | | |
| 3.9.2.11 Object Type Hierarchies | | SQL3 |
| 3.9.2.12 Object Encapsulation | | SQL3 |
| 3.9.2.13 Versions and Configs. | | |
| 3.9.2.14 Archival Storage | | |
| 3.9.2.15 Schema Evolution | Part 1, Section 3.1.5 Not Explicit | Section 11 (Schema Definition and Manipulation) |
| 3.9.2.16 Long Transactions | | |
| 3.9.2.17 Rule Processing | Part 1 Section 5.1.4 SACF Rules | SQL3 |
| 3.9.2.18 Domain-specific Stds | Part 1 Section 5 App. Contexts | Undetermined |

# Bibliography

Bernstein, P., Hadzilacas, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems.* Reading, Massachusetts: Addison-Wesley, 1987.

Biliris, A., Dar, S., Gehani, N., Jagadish, H. V., and Ramamritham, K., "ASSET: A system for supporting extended transactions," in *Proceedings of ACM SIGMOD Conference*, May 1994.

Booch, G., *Object-Oriented Design.* Redwood City, CA: The Benjamin/Cummings Publishing Company, 1991.

Buchmann, A., McCarthy, D., Hsu, M., and U.Dayal, "Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control," in *The Fifth International Conference on Data Engineering*, February 1989.

Carey, M. J., DeWitt, D. J., Richardson, J. E., and Shekita, E. J., *Object-Oriented Concepts, Databases and Applications*, ch. Storage Management for Objects in EXODUS, pp. 341–369. Addison-Wesley Publishing Company, 1989.

Chen, P., "The entity-relationship model - toward a unified view of data," *ACM Transactions on Database Systems*, vol. 1, March 1976.

Date, C. and Darwen, H., *A Guide to SQL Standard.* Reading, MA.: Addison-Wesley Publishing, 1992.

Diaz, O. and Gray, P. M. D., "Semantic-rich user-defined relationship as a main constructor in object-oriented databases," in *Object-Oriented Databases: Analysis,Design & Construction (DS4)*, (Meersman, R., Dent, W., and Khosla, S., eds.), pp. 207 – 224, Elsevier Science Publishers, B.V. (North-Holland), 1991.

DiPippo, L. C. and Wolfe, V. F., "Object-based semantic real-time concurrency control," *Proceedings of the 14th IEEE Real-time Systems Symposium*, December 1993.

DiPippo, L. C., *Object-based semantic real-time concurrency control.* PhD thesis, University of Rhode Island, 1995.

DiPippo, L. C., Wolfe, V. F., and Black, J. K., "Supporting concurrency, timing constraints and imprecision in objects," Technical Report URI-TR94-230, University of Rhode Island, Department of Computer Science, February 1994.

Doherty, M., Peckham, J., and Wolfe, V. F., "Implementing relationships and constraints in an object-oriented database using monitors," in *Proceedings of the 1st International Workshop on Rules in Database Systems*, Springer-Verlag, 30 Aug. - 1 Sept. 1994.

Drew, P. and Pu, C., "Asynchronous consistency restoration under epsilon serializability," Technical Report OGI-CSE-93-004, Department of Computer Science and Engineering, Oregon Graduate Institute, 1993.

Eswaren, K., "Specification, implementation and interactions of a rule subsystem in an integrated database system," Technical Report Report RJ1820, IBM Research, San Jose, CA, August 1976.

Fisher, D., *X3H2-94-488 Real-time Extensions to SQL.* 1994.

Fortier, P., "ANSI DBSSG PRISTG: Real-time database management systems reference model," *ANSI Data Base Systems Study Group Predictable Real-time Information Systems Task Group, PRISTG Document No. 94-001*, January 1994.

Fortier, P., *Early Commit.* PhD thesis, University of Massachusetts Lowell, 1993.

Fortier, P., "A real-time database management systems reference model," *Submitted to ANSI Data Base Systems Study Group (DBSSG) Predictable Real-time Information Systems Task Group (PRISTG)*, June 1994.

Fortier, P. and Sawyer, C. G., "DISWG a new player in NGCR open systems standards," *To appear in Computer Standards and Interfaces*, 1994.

Fortier, P., Wolfe, V. F., and Prichard, J., "Flexible real-time SQL transactions," in *IEEE Real-Time Systems Symposium*, Dec. 1994.

Fortier, P., Wolfe, V. F., and Prichard, J., "RTSQL: Real-time database extensions to the SQL2 standard," 1995. To appear in *Standards and Interface Journal.*

Gallagher, L., "Database management standards: Status and applicability," *Computer Standards and Interfaces*, vol. 12, 1991.

Gallagher, L., "Object SQL: Language extensions for object data management," in *International Society for Mini and Microcomputers Conference on Information and Knowledge Management*, August 1992.

Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., and Salem, K., "Modeling long-running activities as nested sagas," *Bulletin of the IEEE Technical Committee on Data Engineering*, vol. 14, March 1991.

Gordon, K., *DISWG Database Management Systems Requirements.* Alexandria, Virginia: NGCR SPAWAR 331 2B2, 1993.

Herlihy, M. and Wing, J., "Linearizability: a correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, July 1990.

Hughes, D., ed., *Zip_RTDBMS The Real-Time Data Dase Management System.* DBx, Inc., 1993.

Johnson, S. C., "YACC-yet another compiler compiler," Technical Report CSTR 32, Bell Laboratories, Murray Hill, N. J., 1975.

Kligerman, E. and Stoyenko, A., "Real-time Euclid: A language for reliable real-time systems," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 941–949, Sep. 1986.

Korth, H., Levy, E., and Silberschatz, A., "A formal approach to recovery by compensating transactions," in *Proceedings of the 16th Very Large Data Base (VLDB) Conference*, 1990.

Krupp, P., Schafer, A., Thurasingham, B., and Wolfe, V. F., "On real-time extensions to the common object request broker architecture," in *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications (OOPSLA) '94 Workshop on Experiences with the Common Object Request Broker Architecture (CORBA)*, September 1994.

Lesk, M. E., "LEX-a lexical analyzer generator," Technical Report CSTR 39, Bell Laboratories, Murray Hill, N. J., 1975.

Levine, J. R., Mason, T., and Brown, D., *lex & yacc.* Sebastopol, CA: O'Reilly & Associates, Inc., 1995.

Lin, K.-J. and Natarajan, S., "Expressing and maintaining timing constraints in FLEX," in *IEEE Real-Time Systems Symposium*, pp. 96–105, December 1988.

Liu, C. L. and Layland, J. W., "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, pp. 46–61, 1973.

McCarthy, D. R. and Dayal, U., "The architecture of an active database management system," in *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, (Portland Oregon), June 1989.

McKenzie, E. and Snodgrass, R., "Extending the relational algebra to support transaction time," in *Proceedings of the 1987 ACM SIGMOD International Conference on the Management of Data*, (San Francisco, California), May 1987.

Melton, J., ed., *ANSI X3.135-1992, American national Standard, Database Language SQL.* American National Standards Institute, 1992.

Melton, J., ed., *ISO/IEC JTC1/SC21/WG3 DBL YOW-006 and ANSI X3H2-95-086, (ISO/ANSI working draft) SQL Persistent Stored Modules (SQL/PSM).* American National Standards Institute, March 1995.

Melton, J. and Simon, A., *Understanding the New SQL: A Complete Guide.* San Mateo, CA.: Morgan Kauffman Publishers, 1992.

Ng, P. and Paul, J. F., *A formal definition of entity-relationship models.* Amsterdam: North Holland, 1988.

Ozsu, T. and Valduriez, P., *Principles of Distributed Database Systems.* Englewood Cliffs, New Jersey: Prentice Hall Inc., 1991.

Peckham, J., *Constraint Based Analysis of Database Update Propagation.* PhD thesis, University of Connecticut, 1990.

Peckham, J. and Maryanski, F., "Semantic data models," *ACM Computing Surveys*, vol. 20, pp. 153–189, September 1988.

Prichard, J., DiPippo, L. C., Peckham, J., and Wolfe, V. F., "RTSORAC: A real-time object-oriented database model," in *Proceedings of the International Conference on Database and Expert Systems Applications*, September 1994.

Pugh, W. and Marlow, T., eds., *Proceedings of the ACM SIGPLAN workshop on language, compiler and tool support for real-time systems.* ACM SIGPLAN, 1994.

Purimetla, B., Sivasankaran, M., Stankovic, J., Ramamritham, K., and Towsley, D., "Priority assignment in real-time active databases," Technical Report TR94-29, Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610, April 1994.

Ramamritham, K., “Real-time databases,” *International Journal of Distributed and Paralled Databases*, vol. 1, 1993.

Ramamritham, K. and Pu, C., “A formal characterization of epsilon serializability,”. To appear in *IEEE Transactions on Knowledge and Data Engineering.* Also available as technical report No. CUCS-044-91 at Department of Computer Science, Columbia University, 1995.

*ISO/IEC 9579-2, Information Technology, Remote Database Access, Part 2: SQL Specialization.* American National Standards Institute.

Sha, L., *Modular Concurrency Control and Failure Recovery – Consistency, Correctness and Optimality.* PhD thesis, Carnegie-Mellon University, 1985.

Snodgrass, R. and Ahn, I., “Temporal databases,” *IEEE Computer*, vol. 19, pp. 35 – 42, 1986.

Snodgrass, R., et. al., “TSQL2 language specification,” *ACM SIGMOD Record*, vol. 23, pp. 65–86, March 1994.

Son, S., Yannopoulos, S., Kim, Y.-K., and Iannacone, C., “Integration of a database system with real-time kernel for time-critical applications,” in *International Conference on System Integration*, June 1992.

Son, S. H., “Real-time database systems: A new challenge,” *Data Engineering*, vol. 13, December 1990.

Song, X., *Data Temporal Consistency in Hard Real-Time Systems.* PhD thesis, The University of Illinois at Urbana-Champaign, 1992.

Soo, M. D., “Bibliography on temporal databases,” *ACM SIGMOD Record*, vol. 20, pp. 14 – 23, March 1991.

Stachour, P. and Thurasingham, B., “SQL extensions for security assertions,” *Computer Standards & Interfaces*, vol. 11, 1990.

Thurasingham, B. and Schafer, A., “RT-OMT: A real-time object modeling technique for designing real-time database applications,” in *Proceedings of the Second IEEE Workshop on Real-Time Applications*, pp. 124–129, July 1994.

Ulusoy, O., “Current research on real-time databases,” *SIGMOD Record*, vol. 21, pp. 16 – 21, December 1992.

Wells, D., Blakely, J., and Thompson, C., "Architecture of an open object-oriented database management system," *IEEE Computer*, vol. 25, pp. 74 – 83, October 1992.

Wolfe, V. and Cingiser, L. B., "Issues in object-oriented real-time databases," in *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, May 1992.

Wolfe, V., Davidson, S., and Lee, I., "*RTC*: Language support for real-time concurrency," *Real-Time Systems*, vol. 5, pp. 63–87, March 1993.

Yu, P., Wu, K., Lin, K., and S.Son, "On real-time databases: Concurrency control and scheduling," *Proceedings of the IEEE*, vol. 82, January 1994.

Zdonik, S. and Maier, D., *Readings in Object Oriented Database Systems.* San Mateo, CA: Morgan Kauffman, 1990.