RTSQL: Extending the SQL2 Standard to Support Real-Time Databases

J.J. Prichard
Computer Science Dept.
University of Rhode Island
Kingston, RI 02881 USA
prichard@cs.uri.edu

Paul J. Fortier
U. of Massachusetts/Dartmouth
North Dartmouth, MA 02747 USA.
pfortier@umassd.edu

Victor Fay Wolfe
Computer Science Dept.
University of Rhode Island
Kingston, RI 02881 USA
wolfe@cs.uri.edu

Abstract

This paper presents real-time database extensions to the SQL2 standard database language. It 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.

1 Introduction

In 1986 the American National Standards Institute (ANSI) standardized the database language SQL. Since then, there have been efforts to create extensions to the standard for various special purposes including: security [ST90], temporal data [Sno+94], and multi-media data [Gal92]. This paper describes real-time SQL (RTSQL), an extended version of SQL to support *real-time* database systems. Real-time databases are designed to support applications in which the freshness of data and the timeliness of processing are important. Many organizations have contributed to and supported this work, they include: the University of Rhode Island, the University of Massachusetts - Dartmouth, the U.S Department of Defense's Next Generation Computer Resources (NGCR) Database Interface Standard Working Group (DISWG), and the American National Standards Institute's (ANSI) Predictable Real-time Information Management Task Group (PRIS-TG).

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 transactions that determine whether a particular stock trade will generate a profit. These transactions must complete before given deadlines so that decisions can be made on whether to actually initiate a stock trade.

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. In this paper we describe several of these extensions. We first describe how the notion of constraints is extended for both data definition and data manipulation statements. Then we introduce *directives*, which are added language features that express architectural dependencies and resource limitations that are necessary for achieving predictable timing behavior. Finally, we describe the flexible transaction structure introduced by RTSQL. The discussion is high-level and assumes some familiarity with the SQL standard.

This paper is structured as follows. Section 2 provides background information on real-time databases and SQL. Section 3 describes the syntax and semantics of the real-time extensions to SQL. Section 4 summarizes and describes on-going and future related efforts.

2 Background

Standardization efforts for database management systems are relatively new [Gal91]. Among the first major database management systems standards to emerge was the database language NDL for network model database applications in 1986. 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. We provide a brief overview the features of SQL2 in Section 2.1. 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], multimedia (SQL/MM) [Gal92], and remote data access (SQL/RDA) [RDA].

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]. These standards bodies have worked closely to develop recommendations for real-time and next generation database management systems standards. It is these extensions on which this paper is based. We overview issues and related work on real-time database systems in Section 2.2.

2.1 SQL2

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 [Sno+94] 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 Real-Time Databases

A real-time database system has three distinguishing features: the requirement of temporally consistent data, the requirement of timing constraints on execution, and the requirement that certain execution 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 data definition and data manipulation parts of the database query language.

2.2.1 Data Definition in Real-Time Databases

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 temporally 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 would be 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.

Another important aspect of data definition is to provide information that facilitates maintaining high data availability, which in turn can facilitate enforcement of transaction timing constraints. One way to support data availability is to decompose a conventional monolithic database into *database partitions* [For93, HW90, KLS90]. These partitions could be, for instance, atomic data sets, as in [Sha85], or real-time database objects, as in [PDPW94]. This finer grained partitioning of the database allows for looser application of constraints across the partitions than is possible when constraints are applied to the monolithic database. This added flexibility can increase the chances that the database manager can schedule transactions to meet their timing constraints.

2.2.2 Data Manipulation in Real-time Databases

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 paper, 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 [BMHU89]. 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 database partitions must wait to commit until all operations have been performed. 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.

2.2.3 Summary of Real-Time Database Requirements.

In summary, a real-time database system standard should provide support, beyond that of conventional database capabilities, for specifying:

- 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.

A more detailed set of real-time database interface standard requirements can be found in the NGCR DISWG Requirements Document [Gor93].

3 Real-Time Extensions to SQL2

In this section 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 a number of areas. The first area is how the notion of SQL2 *constraints* is extended for both data and execution. The second area is the addition of a new construct called a *directive*. The third area is *transaction structure*. Each of these areas is discussed in the following sections.

3.1 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 on execution specified.

3.1.1 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 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 2.1), 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, the user can use a `GET DIAGNOSTICS` statement to determine which constraint was actually violated, and take 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.

3.1.2 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 3.1.2, 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.

Thus, the user must utilize the timing constraints and condition handlers in a fashion which 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 statement 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.

3.2 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.

Data Dependencies. As mentioned in Section 2.2.1, one method of supporting increased data availability is through database partitioning. These partitions can be determined by examining constraints on data (i.e. a constraint specification mentions more than one data item), and knowledge of dependencies between data objects (i.e. one data item is computed from others). SQL2 provides mechanisms for constraint specification that could be analyzed to determine some of the data dependencies. Data dependencies due to other factors can be specified by the database designer during the definition of a table through the following RTSQL directive:

```
<depends on clause> ::=
  [DEPENDS ON <table name> [WHEN <boolean expression>]
```

For example, referential integrity constraints may specify that the name of a stock appearing in a stock price table must exist in a second table which lists the names of all the stocks available through a particular exchange such as the New York Stock Exchange. The schema could be analyzed to determine that this dependency exists, and as such, that these two tables should exist within the same partition. Furthermore suppose there is another table of mutual funds, which is computationally dependent on the stock price table. This could be indicated by the `<depends on clause>` as follows:

```
CREATE TABLE mutual_funds (fund_name CHARACTER(30), ...)
  DEPENDS ON stock_prices;
```

Once the data dependencies are computed they could then be used to define database partitions. These partitions can then be utilized to determine the possibility of early commit of a subtransaction [For93], which can facilitate meeting transaction timing constraints.

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>
```

Decided that asynchronous execution was not a constraint, but a directive, so here it is...

Asynchronous Execution. In real-time applications, it may be useful to notify the system that some actions can be done aynchronously. 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 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 aynchronous statements to complete execution. If the aynchronous statements have already completed execution when the WAIT statement is executed, an exception condition will be raised.

Worst Case Execution Time. The last addition is a directive for specifying 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[PE94]. 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.

3.3 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].

3.3.1 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 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 transaction includes database access code and transaction processing code. The postcondition contains predicates that define what constitutes a correct execution of

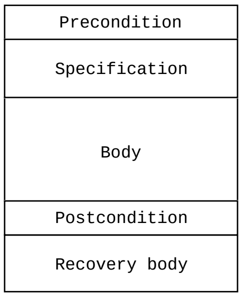

Figure 1: Abstract View of Flexible Transaction Structure

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.

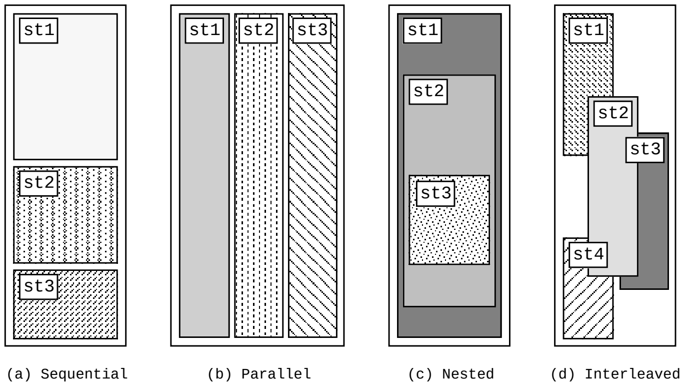

Figure 2: Flexible Subtransaction Structure

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 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 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 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 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].

3.3.2 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.

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

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 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 3.3.1 apply here. Thus entire transactions can also be time constrained.

In SQL2, *datetime* valued expressions have been defined, and can include references to *datetime* value functions. To better support transaction structuring, RTSQL extends the available *datetime* value functions

```
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

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.

3.3.3 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.

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 asynchronous execution

```
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

```
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

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 Conclusion

This paper has described real-time extensions to the ANSI/ISO standard SQL query language. 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 date-time functions, which, along with the timestamps, are sufficient to specify temporal consistency constraints. Database partitioning, which can allow higher data availability, is supported by the addition of a `DEPENDS ON` data definition directive. Timing constraints on general data manipulation statements, including transactions, are specified with a complete set of timing constraints available in a `START TRANSACTION` statement, including: start times, deadlines and periods. 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

```
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

kept in main memory only. Significant extensions to SQL2's transaction capability support the necessity of flexible predetermined execution entities in a real-time database system. 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.

There are two major areas of further work on real-time extensions to SQL2. The first area involves recovery. SQL2 has only primitive recovery support – it can deny operations and provide diagnostics as to why. Recovery needs to be enhanced because of the introduction of full and flexible transaction capabilities, and because of timing and predictability considerations. In particular, full exception handling that allows for forward recovery instead of just traditional database backward recovery is being studied, as are recovery techniques for subtransactions in the various flexible SQL/RT transaction structures that this paper presented. The second major area of study is that of remote data access support in SQL. Since many real-time applications are inherently distributed, the ability for SQL to support data access in distributed sites is important. The DISWG and ANSI PRIS-TG groups are continuing the effort in these and other areas to compose a full SQL/RT standard extension. Comments on the concepts introduced in this paper are welcomed and encouraged; they may be directed to any of the authors.

Acknowledgements. We thank Bhavani Thurasingham, Joan Peckham, Lisa Cingiser DiPippo, Donna Fischer, Karen Gordan, Sang Son, Mike Moore, David Hughes, Jeff Richey, Jim Melton, and the members of the NCGR DISWG and the ANSI PRIS-TG for their ideas, work, and feedback in the development of RTSQL.

References

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

[BMHU89] 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.

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

[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.

[HW90] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[KLS90] H. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the 16th VLDB Conference*, 1990.

[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.

[Mel92] J. Melton, editor. *ANSI X3.135-1992, American national Standard, Database Language SQL*. 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)*. March 1995.

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

[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.

[PE94] W. Pugh and T. Marlow (Editors). *Proceedings of the ACM SIGPLAN workshop on language, compiler and tool support for real-time systems*. ACM SIGPLAN, 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.*

[Sno+94] R. Snodgrass et. al. TSQL2 language specification. *ACM SIGMOD Record*, 23(1):65–86, March 1994.

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

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

[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.