Flexible Real-time SQL Transactions

Paul J. Fortier
U.S. Naval Undersea Warfare Center
Newport, Rhode Island 02841-1708
Fortier@ada.npt.nuwc.navy.mil

Victor Fay Wolfe and J.J. Prichard
Computer Science Department
University of Rhode Island
Name@cs.uri.edu

Abstract

Although significant research has been been done in many facets of real-time database design, there has been little work in real-time transaction structuring and specification. The adherence of conventional transactions to the ACID properties reduces data availability, which reduces the flexibility of real-time transaction scheduling. This paper presents flexible transaction structuring capabilities that allow relaxed ACID properties for better support of real-time transactions. The specification of these flexible transaction structures in a real-time extension to the SQL standard language is described as are simulation results of the flexible transaction structure's effectiveness compared to conventional transactions.

1 Introduction

The need for *real-time database management systems* is becoming more evident as massive data-intensive projects, such as the national information infrastructure (NII) [11], are being developed. NII applications include electronic commerce, digital libraries, advanced manufacturing, environmental monitoring, and health care; all of which require high data availability and may have time-constrained transactions. High data availability improves transaction throughput. It also supports real-time transaction scheduling by removing data contention constraints so there is more flexibility in determining schedules that meet transaction timing constraints. In recent years, there has been a significant amount of research towards developing real-time database models [18, 16], real-time transaction scheduling [1], and real-time concurrency control [21]. However, there has been little work done on structuring and specification of real-time transactions to better support data availability and timing constraints.

Conventional transaction structures and specification techniques are not adequate for real-time transactions for several reasons. First, conventional techniques do not support timing constraints on transaction execution. Second, unlike conventional transactions that are typically assumed to be independent, real-time applications often require that transactions synchronize with each other to support integrated real-time control algorithms. This synchronization may be between concurrent transactions, nested transactions, parallel transactions, or combinations of these structures. Third, real-time transactions may need to support non-traditional notions of consistency, such as partial atomicity or relaxed serializability. Finally, real-time transactions have to support non-traditional recovery techniques that are better suited to real-time systems. To support these additional requirements, new transaction structures and specification languages must be developed.

For the past two years there has been a coordinated effort among the U.S Department of Defense's Next Generation Computer Resources (NGCR) Database Interface Standard Working Group (DISWG), the American National Standards Institute's (ANSI) Predictable Real-time Information Management Task Group (PRIS-TG), researchers at the Naval Undersea Warfare Center and researchers at the University of Rhode Island to develop real-time extensions to the standard SQL database data definition/data manipulation language. Part of this effort has focused on extending SQL to support the specification of real-time transactions. This paper describes proposed extensions to SQL and simulation results for real-time transaction structuring that have been developed.

2 Related Research

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). Most efforts in real-time database research involve redefining one or more of these terms. For instance, consistency may be redefined by relaxing serializability [21] and durability may be be redefined by taking into account *temporal consistency* of data [18]. The conventional model of transaction recovery used to enforce the ACID properties is also not well-suited to real-time transaction processing. Recovery based on transaction save points and undo and redo looses sight of the dynamic forward motion of *time* in a real-time database. Recovery in real-time database management systems must take into account applications needs when determining if and how to recover [7, 20].

Real-time transactions are characterized in [18] along three dimensions. First is the way in which the data is used by the transaction: *sensor* (write-only) transactions obtain the state of the environment and periodically write the sensed data to the database; *update* transactions both read from and write to the database either periodically or aperiodically; and *read-only* transactions, such as user queries, read data from the database, also either periodically or aperiodically. The second dimension is the nature of the timing constraints. Timing constraints on transactions can be imposed by the temporal consistency requirements of the data that is written. That is, the periodicity requirement of a transaction may be derived from the absolute validity interval of the data that it writes. A transaction's timing constraints can also come from requirements imposed on system reaction time. The third dimension along which transactions are characterized considers the implication of a missed deadline. Some transactions are considered hard transactions which may result in catastrophe if a deadline is missed. Such transactions must be guaranteed to meet timing constraints. Other transactions are soft transactions which have some diminished value after a missed deadline.

Transaction scheduling in a database system implies both transaction selection and concurrency control, which is a more complicated than typical task scheduling in an operating system [1, 13]. Concurrency control is one of the most widely studied areas in real-time databases because many researchers have recognized the inherent conflict between meeting timing constraints and maintaining database consistency (for a survey of real-time concurrency control and scheduling techniques see [21]).

To date, there has been little work on real-time transaction structuring and specification. In particular, although database query languages have been evolving over the last few years, there has been little effort towards incorporating real-time features. Recently, practitioners have seen the merits of standards in various areas of computer systems. ANSI has standardized the SQL2 language. Beyond this, the SQL3 evolving standardization effort is seeking to incorporate the object model [8]. Efforts by ANSI's PRIS-TG and NGCR's DISWG groups are formulating real-time recommendations for SQL [6]. It is these efforts from which the results in this paper are derived.

3 Flexible Transaction Structure

Real-time database management requires a new model for database and transaction structure that better supports data availability and timing requirements. One way to support data availability is to decompose a traditional monolithic database into *database partitions*. These partitions could be, for instance, atomic data sets, as in [19], or real-time database objects, as in [16]. 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. Transactions too must be restructured for real-time databases. In this section, we discuss transaction structure including relaxation of the traditional ACID properties of large transactions through the decomposition to subtransactions.

3.1 Transaction Requirements

Traditional transactions are structured to enforce the ACID properties. The *isolation* ACID requirement causes operations that depend on each other to be grouped into a single transaction so that enforcement of the ACID properties maintains the dependencies. However, the inflexibility of this structure often causes conservative execution of transactions that reduces data availability. 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 traditional 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, we propose a redefinition of the ACID requirements that allow for more flexibility and more data availability. 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 redefinition of ACID properties for real-time transactions includes the following:

- *Atomic* - An atomic transaction implies the all or nothing execution of the transaction. For flexible real-time transactions, atomic execution is selectively applied to these pieces of the transaction that have a critical need for totally consistent data. The atomic property is applied to these pieces of the transaction acting on disjoint database partitions to ensure consistency on the decomposed database partitions in isolation.

- *Consistent* - Transactions given an initial consistent monolithic database state will transform the database to a new consistent state. In this traditional definition, consistency is determined by the evaluation of constraints defined on the entire database. In real-time databases certain forms of consistency may have to be relaxed to better support timing constraint maintenance. Also, consistency may be enforced on database partitions rather than on the whole database. This definition of consistency allows for incremental checking and validating thereby increasing data availability, potential concurrency, and lessening transaction failure due to constraint violations [4, 5, 15, 19].

- *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 between transactions. In real-time databases, transactions may need to communicate and synchronize with other transactions to perform control functions. Transactions may need to synchronize on external time boundaries, system events, another transaction's results or end conditions, or they may need to perform some integrated set of tasks for an application that requires sharing of system state knowledge.

- *Durable* - The durability property of transactions implies that the results of a transaction are persistent and permanent. In a real-time database system, this property may not hold uniformly; temporal consistency may be used instead. That is, data may be assigned a lifetime after which it becomes nonpersistent, or a data item may be part of a database structure that has events at which time data items disappear. For example, a circular queue data structure, which is commonly used in real-time databases, requires deletion of the oldest item when the queue is full and a new item is added. Durability of data must be specified semantically by the constraints and structure of the data, not by an implicit feature of a database or by a transaction's execution.

- *Correct* - We add a fifth property to those of traditional transactions: Correctness. Correctness is defined by transaction *preconditions* and *postconditions*, which can specify a rich set of user-defined semantic correctness. Included is the specification of transaction timing constraints such as start times, finish times, and period frames.

3.2 Transaction Structure

Our real-time transaction structure, which is designed to support the above requirements, includes: a *precondition part*, a *specification part*, a *body*, a *postcondition part*, and a *recovery* part (Figure 1). The precondition consists of requirements which 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 [6]. The body of the transaction

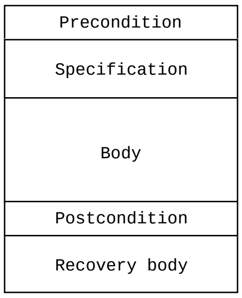

Figure 1: Abstract View of Flexible Transaction Structure

includes database access code and transaction processing code. The postcondition contains predicates which 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.

To facilitate the specification of application execution models that best support data availability, our transaction model includes constructs for partitioning transactions into subtransactions within the internal structure of the transaction body. A subtransaction can have all of the components that we have defined for a transaction, but in addition a subtransaction can include information on the type of subtransaction processing supported. There are four basic structural building blocks for

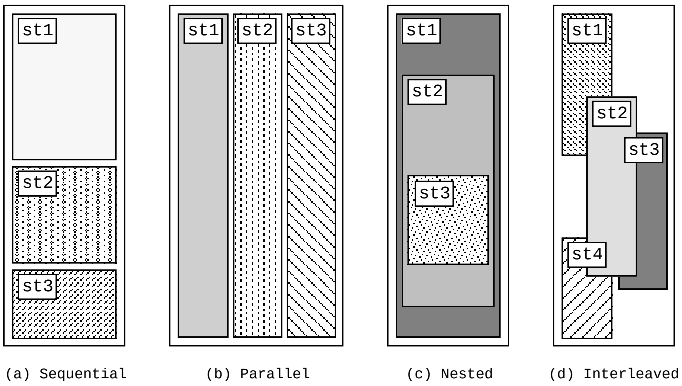

Figure 2: Flexible Subtransaction Structure

transactions: 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 [10].

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. Ths 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.

4 Real-time SQL2 Language Specification

The SQL2 standard database language [3] consists of three subparts: a data definition part, a data manipulation part and a data control part. The data definition language is used to define the structure and constraints of the database schema. The data manipulation language is used to manipulate the data defined by the data definition language specification. The data control language is used for managing access to the database. In the following subsections, we propose a number of extensions to SQL2 to support the structure of flexible real-time transactions.

4.1 Database Specification

As mentioned in Section 3, one method of supporting increased data availability is through database partitioning. These partitions can be determined by examining constraints on data (i.e. the constraint mentions more than one data item), and knowledge of dependencies between data objects (i.e. one data item is computed from others). SQL2 constraint specifications could be analyzed to determine some of the data dependencies. Data dependencies due to other factors can be specified during the definition of a data item through a `DEPENDS ON` clause. These database partitions could then be considered when determining the possibility of early commit of a subtransaction. How and where the data is stored by the system can also have an affect on transaction processing and timely data availability [6].

4.2 Transaction Specification

To specify the transaction structuring of Section 3, mechanisms and constraint definitions that allow for control over transaction execution and commit must be defined. The transaction structure in Section 3.2 described five parts of a transaction: a precondition, a postcondition, a specification, a body, and a recovery part. Since SQL2 does not provide any mechanism for writing named transactions, the following syntax is proposed:

```
<transaction-specification> ::=
  TRANSACTION <transaction name> (<input parameters>)
    [<local declarations>]
    [[REPLACEABLE] PRECONDITION <boolean exp>]
    [[REPLACEABLE] POSTCONDITION <boolean exp>]
    [[REPLACEABLE] RECOVER ON <condition>
      [AUTO | SEMI <transaction name> | MANUAL]]
    [[REPLACEABLE] CRITICALITY_LEVEL <criticality-level>]
    [[REPLACEABLE] ISOLATION_LEVEL <isolation-level>]
    [[REPLACEABLE] ACCESS_MODE <access-mode>]
    [[REPLACEABLE] DIAGNOSTICS SIZE <value spec>]
  BEGIN
    <transaction body>
    //A COMMIT or ROLLBACK statement must terminate execution of the transaction
  END
```

The PRECONDITION clause allows specification of a predicate which must evaluate to true before the transaction may begin execution. The POSTCONDITION clause allows specification of a predicate which 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 CRITICALITY_LEVEL clause specifies a relative criticality level for this transaction, which can be used by the underlying system in determining the transaction execution schedule. 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, 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 [9].

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 would be modified as follows:

```
<set transaction> ::=
SET TRANSACTION <transaction name>
    [PRECONDITION  [AND | OR] <boolean exp>]
    [POSTCONDITION [AND | OR] <boolean exp>]
    [RECOVER ON <condition>
      [AUTO | SEMI <transaction name> | MANUAL]]
    [CRITICALITY_LEVEL <criticality-level>]
    [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

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

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, the following syntax is proposed:

```
BEGIN TRANSACTION <transaction name> (<input arguments>)
  [SYNC | ASYNC]
  [START BEFORE <datetime value exp>]
  [START AFTER  <datetime value exp>]
  [COMPLETE BEFORE <datetime value exp>]
  [COMPLETE AFTER  <datetime value exp>]
  [PERIOD <interval value exp>
    [START AT <time exp>]
    [UNTIL <boolean exp>]]

<datetime value function> ::=
  CURRENT_DATE | CURRENT_TIME[<time precision>] |
  CURRENT_TIMESTAMP[<timestamp precision>] |
  EINITIATE(<tid) | ESTART(<tid) |
  ECOMPLETE(<tid>) | ECOMMIT(<tid>) | EABORT(<tid>)
```

When the BEGIN 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*) which 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.

The first clause of the BEGIN TRANSACTION statement specifies whether the transaction is to run synchronously or asynchronously. If left unspecified, the default is synchronous. The BEGIN TRANSACTION statement also allows the specification of timing constraints on the transaction execution. In SQL2, *datetime* valued expressions have been defined, and can include references to *datetime* value functions. We have extended the available *datetime* value functions to include EINITIATE, ESTART, ECOMPLETE, ECOMMIT, and EABORT. 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 would need to be extended to accommodate this value appropriately.

```
TRANSACTION main ()
  tid st1, st2, st3;
  POSTCONDITION NOT REPLACEABLE
    (((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 := BEGIN TRANSACTION trans1();
  st2 := BEGIN TRANSACTION trans2();
  ...
  st3 := BEGIN TRANSACTION trans3();
  COMMIT;
END
```

Example 1: Sequential Transactions

4.3 Examples of RT-SQL Flexible Transaction Structures

With these extensions, SQL2 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 := BEGIN TRANSACTION trans2();
  ...
  COMMIT;
END

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

```
TRANSACTION trans2()
  tid st;
BEGIN
  ...
  st := BEGIN 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

```
TRANSACTION main ()
  tid st1, st2, st3;
  REPLACEABLE POSTCONDITION ((ECOMMIT(st1) <> UNDEFINED) OR (ECOMMIT(st3) <> UNDEFINED))
BEGIN
  st1 := BEGIN ASYNC TRANSACTION trans1();
  st2 := BEGIN ASYNC TRANSACTION trans2()
           START AFTER ESTART(st1)
           START BEFORE ECOMPLETE(st1);
  st3 := BEGIN 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

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` which specifies that the transaction should start within 2 minutes of when the `BEGIN 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.

To model parallel execution of subtransactions requires the addition of a parallel block. SQL3 has block structure defined, but a special construct such as the following should be added for parallel execution:

```
BEGIN PARALLEL
  <transaction initiations>
END PARALLEL
```

Example 4 shows three subtransactions which are to execute in parallel.

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

Example 4: Parallel Transactions

5 Flexible Transaction Concept Validation

In this section, we simulate various models of flexible SQL transactions to demonstrate the improvements in performance that are possible. We examine the performance of flexible transactions when early commit of subtransactions is allowed, which is when sibling subtransactions have no commit dependencies between them. We describe simulation results for four subtransaction execution models: serial subtransactions, nested subtransactions, interleaved subtransactions and parallel subtransactions.

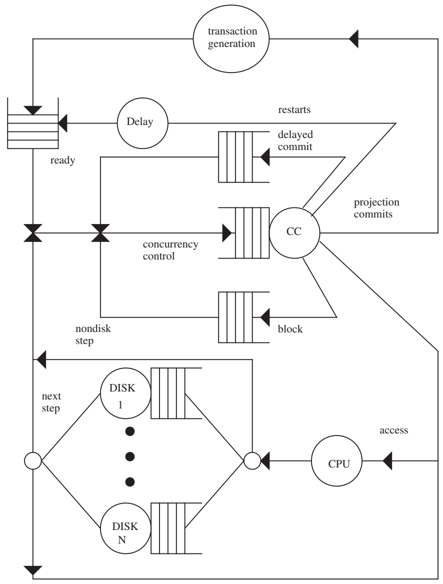

Figure 3: Simulator Queueing Model

Our simulator is designed to follow the simulators described in [2] using the SLAM II simulation language's [17] network and discrete event capabilities. The simulator consists of five distinct service elements with well defined interfaces. These are the transaction generator, the concurrency control server, the commit delay server, the CPU server and the disk server. The transaction generator is responsible for generating transactions based on inputed transaction structural data. The concurrency control server models the actions of executing database operations on the database. We implemented concurrency control versions based on locking, time stamp ordering, and optimistic protocols. The CPU server models the delays associated with the execution of read, write, commit and abort operations. The disk server models the delay associated with accessing a data item from a disk.

The logical queuing model, shown in Figure 3, represents a closed queuing model of a single site database system. We vary the multiprogramming level by infusing additional transactions into the model as needed. As transactions exit the system, new transactions are created to maintain the selected multiprogramming level. Each transaction is modeled as a bounded sequence of database read and write operations for the conventional database protocols, and as bounded subtransactions which have dependencies between them based on the execution model being examined. Subtransactions consist of database read and write operations with commit dependencies noted. 30% of all transaction instructions are writes. Transactions commit when they are ready to do so. If conflicts arise, then transactions are aborted and restarted after a delay. Subtransactions may not abort if not dependent on a failed sibling. The sibling(s) that failed will abort and restart.

5.1 Simulation Results

The simulation measures the performance improvement of flexible transaction and database decomposition constructs against conventional database structuring, transaction structuring and transaction execution schemes. It varies the multiprogramming level from 5 transactions up to 50 transactions for all protocols. The database is composed of 1000 data items partitioned equally over 25 consistency-preserving partitions.

5.2 Nested Subtransactions

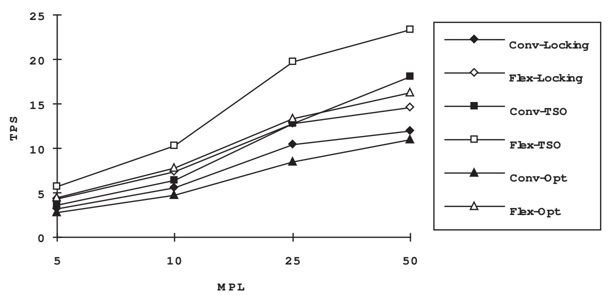

Figure 4: Comparison of Conventional and Nested Flexible Subtransactions

The test in Figure 4 examines how transactions formed as collections of nested dependent subtransactions could improve performance over conventional transaction structures. The average size of a conventional transaction is nine operations and the average size of the flexible transactions is three nested subtransactions with each averaging three instructions. Transaction size varies from three up to twelve operations and from two up to five subtransactions. Subtransactions are nested such that if five subtransactions compose a transaction, then there is a maximum of two nested subtransactions composed of three and two dependent subtransactions in each. In Figure 4, and all subsequent figures, we show the transactions per second (TPS) plotted against the multiprogramming level (MPL) for various concurrency control schemes: conventional transactions with locking (Conv-Locking), flexible transactions with locking (Flex-Locking), conventional transactions with timestamp protocols (Conv-TSO), flexible transactions with timestamp protocols (Flex-TSO), conventional transactions with optimistic concurrency control (Conv-Opt), and flexible transactions with optimistic concurrency control (Flex-Opt).

As the multiprogramming level increases, the performance of the flexible transactions improves compared to the conventional transactions. The maximum improvement is from a low of 17% to 24% for flexible locking. The range is 21% to 38% for flexible optimistic concurrency control. Flexible timestamp ordering ranges from 18% to 37% better than the conventional timestamp protocol. Nested subtransactions appear to suffer from added blocking of children subtransactions which cannot release resources until the parent sibling commits.

5.3 Serial Subtransactions

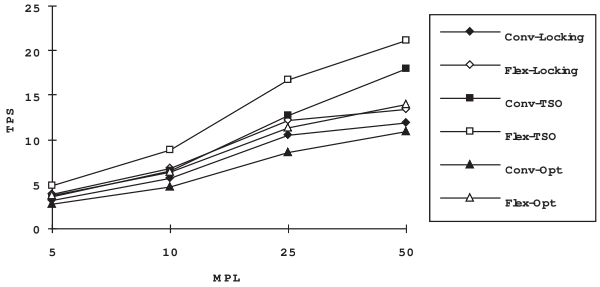

Figure 5: Comparison of Conventional and Serial Flexible Subtransactions

The test of Figure 5 examines how transactions formed as collections of serial nondependent subtransactions improve performance over conventional transaction structures. The average size of a conventional transaction is nine operations and the average size of the flexible transactions is three serial subtransactions with three instructions in each on the average. Transaction size varies from three up to twelve operations and from two up to five subtransactions.

The results of this test are similar to those of the nested subtransaction model. Locking has the lowest gains of from 21% to 38%, timestamp ordering gained from 32% to 51%, and optimistic gained from 49% to 65% over conventional transactions. The improvement in performance over nested subtransactions appears to be due to lessened commit dependencies and restarts. As a sibling reachs its commit point, it commits, which signals its successor to start. Commit of the subtransactions releases resources immediately, whereas in the nested case, release is still delayed until the dependent sibling is prepared to commit.

5.4 Interleaved Subtransactions

The test of Figure 6, with the interleaved subtransaction model, shows improved performance of the flexible transaction from 28% to 50% for locking, from 41% to 79% for time stamp ordering and

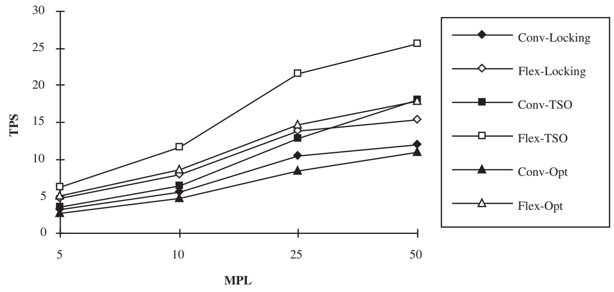

Figure 6: Comparison of Conventional and Interleaved Flexible Subtransactions

from 63% to 83% for optimistic protocols over their conventional counterparts. This result holds over the entire range of transaction multiprogramming levels. The improvement is due to fewer subtransaction dependencies, early commit of subtransactions and the isolation effect of aborts and restarts on individual subtransactions instead of an entire transaction.

The results of this test indicate that as commit dependencies are lessened, even with initiation (precondition) dependencies in place, performance is improved. The earlier release of data items causes further reductions in blocking and restarts of subtransactions, thus allowing for greater data availability and therefore higher concurrency.

5.5 Parallel Subtransactions

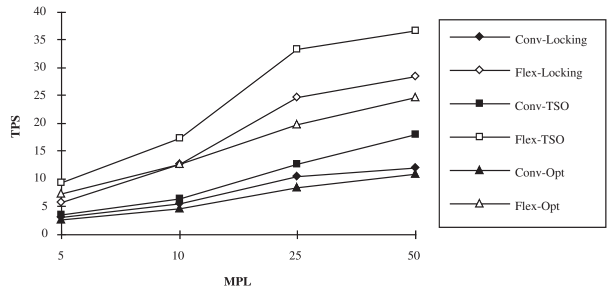

Figure 7: Comparison of Conventional and Parallel Flexible Subtransactions

The test of Figure 7 models transactions as parallel subtransactions. There are no commit dependencies, but all must begin at the same time. Previous simulation parameters are all held constant. Transactions average three subtransactions with three instructions in each. Conventional transactions average nine instructions.

Flexible locking improves over conventional locking from 80% up to 135% for high multiprogramming levels. For the flexible timestamp ordering protocol, the performance improvement ranges from 150% up to almost 200% in comparison to conventional timestamp ordering. The flexible optimistic protocol concurrent executions improve performance from a low of 125% up to 175% over conventional optimistic.

5.6 Database Partitioning

The test of Figure 8 examines the three flexible transaction protocols using the interleaved subtransaction model for different numbers of database partitions. The multiprogramming level is fixed at 25 and the size of the database is fixed at one thousand data items, with the number of data objects divided evenly between the available database partitions. All other simulator parameters are kept constant as defined in the previous tests.

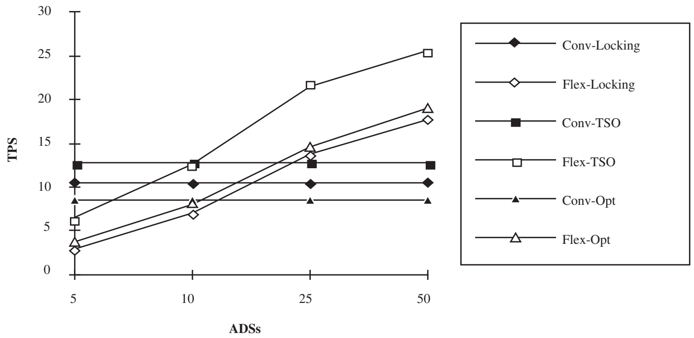

Figure 8: Database Partitioning Sensitivity

The results show that the performance of the three flexible transaction protocols improves as the number of database partitions is increased, although more slowly as the number of partitions approaches fifty. In our initial simulation, database partitions could not span disks, so that decreasing the number of database partitions decreases the number of disks available to the transactions.

The conventional concurrency control protocols out perform the flexible transaction protocols when the number of database partitions are low. However, we believe that this is a result of restricting each database partition to reside on a single disk. We believe that the flexible transaction protocols would approximate that of the conventional protocols for small numbers of database partitions (5 or less), and would perform worse at lower levels due to the increased overhead of the flexible transaction protocols.

6 Conclusions

This paper has examined concepts for flexible transactions for real-time database management specified in the standard SQL2 language. The use of commit dependencies and data dependencies defined between subtransactions makes the flexible subtransactions different from conventional rigid transaction partitioning schemes. These dependencies define how and when commit can occur. Control over these specifications by the programmer allows for early commit or deferred commit that can increase concurrency and data availability. In conventional nested models, sequential models, and parallel models, siblings defer until the parent transaction determines it is commit time. Flexible subtransactions can make this decision explicitly based on their semantic knowledge of the execution dependencies.

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. 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. One such approach requires partitioning the database using consistency contraints so that consistency is enforced on these partitions rather than on the monolithic database. Transactions and subtransactions must then be structured in such that they do not violate the consistency of the partitions. The correctness of such a scheme has been proven by numerous researchers in different environments [5, 12, 14, 19].

The simulation results verify that improvements can be realized if transactions are decomposed into subtransactions whose commit dependencies are specified in pre and postconditions. Providing concurrent execution of non-dependent sibling subtransactions results in earlier commit of subtransactions and higher availability of data. Increased performance is due to shorter average transaction execution time, earlier availability of partitioned data items and reduced data conflict due to early release.

This paper has focused on transaction structuring proposed for RT-SQL. Other RT-SQL real-time features are being developed as well, including: real-time database structure specification, timing constraint specification and enforcement, and directives to bound resource requirements [6].

Acknowledgements. We thank John Sieg for his collaboration in developing the transaction structure and simulation. We also thank the members of ANSI PRIS-TG and NGCR's DISWG groups along with David Hughes of DBx Inc for their efforts in the development of RT-SQL. We thank Lisa Cingiser DiPippo and Joan Peckham for their support in the development of this paper.

References

[1] Robert Abbott and Hector Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *14th VLDB Conference*, August 1988.

[2] R. Agrawal, M. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and applications. *ACM Transactions on Database Systems*, 12(4), December 1987.

[3] *ANSI X3.135-1992, American National Standard for Information Systems - Database Language SQL.* American National Standards Institute, New York, NY, 1992.

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

[5] P. Fortier. *D.Sc. Thesis: Early Commit.* University of Massachusetts Lowell, 1993.

[6] P. Fortier and J.J. Prichard. Concepts for a real-time structured database query language (RT-SQL). *To Appear in the Proceedings of the IFIP/IFAC Workshop on Real-time Programming*, June 1994.

[7] P. Fortier and J. Sieg. Recovery protocols for real-time database management systems. *To Appear in the Proceedings of the International Conference on Information Management (ICIM94*, May 1994.

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

[9] L. Gallagher. Object SQL: Language extentions for object data management. *International Society for Mini and Microcomputers CIKM-92*, 1992.

[10] 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.

[11] IITA Task Group, editor. *ANSI X3.135-1992, American National Standard for Information Systems - Database Language SQL.* National Coordination Office for HPCC, Executive Office of the President, February 1994.

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

[13] Jiandong Huang, John Stankovic, D. Towsley, and Krithi Ramamritham. Experimental evaluation of real-time transaction processing. In *IEEE Real-Time Systems Symposium*, December 1989.

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

[15] H. Korth and G. Speegle. Formal model of correctness without serializability. In *Proceedings of ACM SIGMOD*, pages 379–388, June 1988.

[16] Joan Peckham, Victor Fay Wolfe, J.J. Prichard, and Lisa Cingiser DiPippo. RTSORAC: Design of a real-time object-oriented database system. Technical Report URI-TR-94-231, University of Rhode Island Dept. of Computer Science, March 1994.

[17] A. Pritsker and C. Pegden. *Introduction to Simulation and SLAM.* John Wiley and Sons, inc., New York, NY., 1984.

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

[19] L. Sha. *Ph.D Thesis: Modular Concurrency Control and Failure Recovery – Consistency, Correctness and Optimality.* Carnegie-Mellon University, Pittsburg, PA, 1985.

[20] H. Tokuda. Compensatable atomic actions in object-oriented operating systems. In *Proceedings of the Pacific Computer Communications Symposium*, October 1985.

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