A Parallel Execution Model for Updating Temporal Databases

Avigdor Gal, Opher Etzion*
Technion - Israel Institute of Technology
Faculty of Industrial Engineering and Management
Information Systems Department
Haifa, 32000, Israel
Email: {avigal, ieretzn}@ie.technion.ac.il

Abstract

A parallel execution model for the update process of temporal databases is introduced in this paper, based on temporal parallelism and temporal independence. The parallel approach improves the throughput of massive and complex updates of a multi-version schema system. The notion of temporal agents and its effect on parallelism is discussed, as well as different transaction modes. Several simulation results that present the benefits of the parallel execution model are introduced and discussed.

keywords: temporal databases, parallel processing, schema versioning.

*The work of this author was supported by the H. Kieval Research Fund.

1 Introduction and motivation

Current studies of many contemporary applications in areas such as decision support systems and decision analysis systems show the necessity of representing a time dependent information and of update operations that refer to past and future information [31]. These requirements is partially satisfied by the infrastructure that has been devised within the temporal database area [29], basically aimed at incorporating time concepts as database's primitives. Other works (e.g., [10]) extended the temporal database capabilities to support retroactive and proactive updates, i.e., modifications of information that is valid in past or future time points. For example, a rank awarded to an employee in June 1994 is effective as of January 1994, generating as a result, a **retroactive update**.

Adding time dependent information load databases with massive and complex updates. It has been claimed [30] that the size of information volumes inserted to databases steadily increases every year. Consequently, "the I/O bottleneck worsens" [30], i.e., the response time of updates may become unreasonable and the system's throughput is severely damaged. The I/O bottleneck problem becomes more acute in applications that require the support of temporal databases, since temporal databases should maintain data consistency within the entire application time domain, i.e., all the time points that are referred by an application.

One of the possible solutions to this problem is the use of parallel processing for database updates. Consequently, there is a growing need for tools that take advantage of parallel database systems [6]. In this paper we present an execution model that is based on parallel processing as a solution for the performance problem in temporal database updates. The execution model supports all the update execution model features in conventional databases, and additional three features, as follows.

Bounded temporal effect: The execution model supports update operations that affect a bounded time segment, rather than the entire application time domain. For example, consider an update operation received on July 1994, which reports an employee's promotion to a new temporary rank for a year. The database updates the new rank value to be valid only during [July 1994, July 1995).

Updates throughout the database history: Update operations can relate to time points other than the time point in which the update is initiated. In the example given above, if the employee's promotion is effective only **as of January 1995**, the database proactively updates the new rank's value to be valid during [Jan 1995, Jan 1996). In many temporal databases (e.g., [7]) this requirement is omitted

since past information cannot be altered and future information cannot be modeled.

Schema versioning considerations: The ability of temporal databases to respond to changes in the modeled reality stems from its capability to support meta-data (schema) changes in addition to data changes. A database accommodates *schema versioning* if it supports modifications to the database schema, as well as update and retrieval operations that should be interpreted in the context of the appropriate schema version, which is not necessarily the current one. The persistence of all schema versions guarantees correct interpretation of historical data.

The implementation of temporal updates as atomic entities in a database that supports schema versioning may result in problems such as the one demonstrated in the following simple example. Zipcodes are represented by numeric values during the time interval $[\tau_1, \tau_3)$ and by alphanumeric values during the time interval $[\tau_3, \tau_5)$. Assuming $\tau_1 < \tau_2 < \tau_3 < \tau_4 < \tau_5$, an update operation which assigns the value "1w2zk" to a zipcode during the time interval $[\tau_2, \tau_4)$ can succeed only during $[\tau_3, \tau_4)$. If this update operation is encapsulated within a single atomic transaction, this transaction should abort. The user, who may not be acquainted with the different schema versions, has to figure out manually the reason for this failure and initiate another transaction that would assign this value only in $[\tau_3, \tau_4)$.

The execution model for a temporal database introduced in this paper is based on the following two prominent principles.

Temporal parallelism: Each transaction in a temporal database may consist of updates that refer to different time points. Thus, a user transaction that is temporally partitioned into sub-transactions can be executed in parallel without any loss of integrity.

Temporal independence: Operations that refer to different time points may be partitioned into separate transactions with several possible levels of dependencies among them. This observation enables the support of *temporal independence*, where the temporal database can be viewed as a set of independent database snapshots, each of which relates to a different time point. A user transaction is viewed as a collection of subtransactions applied to different snapshots. Each subtransaction is implemented as a conventional database transaction, with possible mutual commit and abort dependencies.

The execution model utilizes the notion of *information agents* [21]. An *information agent* is a robust, situated, embodied, goal-driven application program. In our context, information agents interpret the updates

operations according to the valid schema at a given time point. The schema evolution results in different interpretations of update operations for different time points, resulting in several information agents.

The main characteristics of the execution model are as follows. The application time domain is partitioned into temporal regions. Each temporal region consists of all the time points during which a single schema is valid. A temporal region is processed by a unique *temporal agent*, which is an application program that uses the schema to interpret update operations and enforce integrity constraints in its temporal region. Modifications to the schema are persisted in the database, and each transaction is partitioned between parallel processes, where each process is executed by a temporal agent. Conflicts are avoided by a coordination protocol among the temporal agents. The transaction model is subject to decisions about the level of possible parallelism and uses a control model that executes parallel subtransactions.

The rest of the paper is organized as follows. Section 2 describes the data model of a temporal database, based on [9]. Section 3 discusses the schema evolution control and the execution model. In Section 4 we present the benefits of the execution model using simulation, and discuss the simulation results. Section 5 provides an overview of related work. The paper is concluded in Section 6.

2 The temporal data model

In recent papers ([9], [13]) we have examined the properties of a general temporal database model as a key technology for supporting complex applications. In this section we briefly describe the main properties of the underlying architecture for the execution model.

2.1 Time types

As argued in [27], temporal functionality in a database is supported using several types of time associated with each data item. A *bitemporal* database [15] consists of two time types, as follows.

Valid time (t_v) - designates the time points at which a data item is considered to be true in the modeled reality. The valid time is expressed using a *temporal element* [11], which is a finite union of time intervals.

Transaction time (t_x) - designates the time when a data item becomes current in the database. This time type may be implemented by using a transaction commit time.

For the support of complex applications, we add a third time type that specifies the time of the real-world event(s) that lead to the transactions that affect the temporal database. We refer to this time type as a *decision time*.

Decision time (t_d) - The time at which a data item's value was decided in the database's domain of discourse [9]. This time point denotes the time at which an event occurred, or the time at which a decision was made. From the database point of view, t_d reflects the time point where an occurrence in the modeled reality entails a decision to initiate a database update transaction. For example, if a person's address was changed on November 12 1991, and the information was inserted to the database on November 13 1991, t_d would be November 12 1991, and t_x would be November 13 1991. The decision time represents the correct order of events in the real world that is not always identical to the order of transaction times.

2.2 The data model

The data model employed in this paper can augment any data model that supports operations at the attribute level. In particular, one may assume an object-oriented model, in which each class has its properties, or a relational model, in which each relation has its columns. The term *class* defines either a class in the object-oriented model, or a relation in the relational model. The term *object* defines an instance of a class or a tuple in a relation. *Property* defines an attribute in the object-oriented model and a column in the relational model. The flexibility in choosing the data model results in a *data model independence*, where the temporal effect can be added on top of existing systems. Data model independence is particularly useful in heterogeneous environments that include legacy systems.

Figure 1 presents a partial schema of a temporal database application that consists of two classes. The class Person represents personal details of a person, and the class Meta-Data-Changes is a meta-data class that stores changes made in the schema level. The underlined properties (e.g., Social-Security-Number) are the object-identifiers, i.e., the properties according to which the object is identified.

A *variable* is an instance of a property in the sense that it is associated with a specific object, an instance of its class. A variable in a temporal database has a set of one or more values, each of which is valid in a validity time interval. A non-temporal variable can be viewed as a special case of a temporal variable. Thus, we assume that each variable has the temporal characteristics.

class= Person
properties= Social-Security-Number
Name
Address:
House-Number
Street
City
Zipcode
State

class= Meta-Data-Changes
properties= Change-Code-Number
Property-Name
Property-Structure

Figure 1: A partial schema example

Address.Zipcode=
(s1) 12345, Dec 1 1990, Dec 1 1990, $[Dec$ 1 1990, $\infty]$
(s2) 12445, Nov 13 1991, Nov 12 1991, $[Jan$ 1 1992, $\infty]$
(s3) 1w2zk, $\underbrace{June\ 5\ 1993,}_{t_x}$ $\underbrace{June\ 4\ 1993,}_{t_d}$ $\underbrace{[Sep\ 1\ 1991,\ \infty]}_{t_v}$

Figure 2: State-elements of John Doe's Zipcode

A variable with temporal extension consists of a variable-state set $\{se_1, \ldots, se_n\}$ of n state-elements. A *state-element* represents the variable's value along with the value's time stamp, which consists of the time types, as presented in Section 2.1. For example, Figure 2 illustrates the changes to the state of the variable Zipcode of John Doe's address. The term ∞ in a t_v of a state-element means that the state-element should hold forever, yet other state-elements can be added with intersecting time intervals. The state-elements of Figure 2 are presented graphically in Figure 3 on a valid time vs. a decision time axis.

3 The update process

In this section we present an update process of a temporal database that enables the generation of new state-elements that may be valid in the past or in the future, as well as in the present. Consequently, several values of the same variable are available and supported at the same time. Section 3.1 presents the general framework of temporal agents, and the effect of schema evolution in this framework. Section 3.2 discusses briefly the update process of a single temporal agent. Section 3.3 describes the process of partitioning update operations and possibly allocating them to other agents. Different consistency modes among agents are presented in Section 3.4.

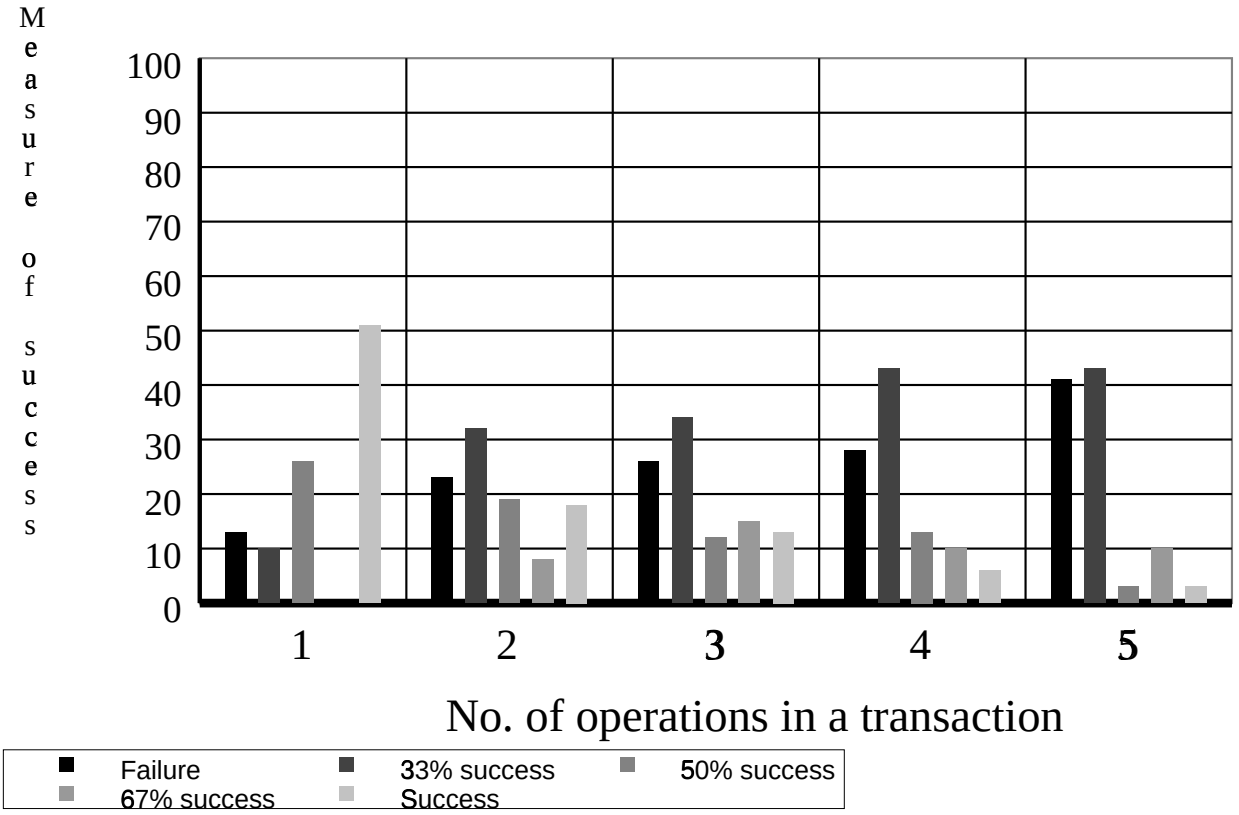

Figure 3: Graphical presentation of the state-elements of Figure 2

3.1 Schema evolution in temporal databases

Let DB be a database of an application with an application time domain ATD. The application time domain is the collection of time-points in which there is a value that is relevant to the application. ATD is initialized to the valid time of the first schema of the application and can be extended later. $DB = D \cup MD$ consists of the data (D) and the meta-data (MD) of the application.
The meta-data of an application $MD = IS \cup IC$ consists of the initial schema (IS) and a set of incremental schema changes ($IC = ic_1, \ldots, ic_n$) that were made to the initial schema. In the example given in Figure 1, IC is represented as a Meta-Data-Changes class.

Let si be a schema description. If i=0 then $si = IS$. Otherwise, si is the schema generated by applying the changes $ic_1, \ldots, ic_i$ on IS. For example, if the initial schema IS contains the property House-Number whose domain is numeric with 2 digits and ic_1 changes the domain of this property to be alphanumeric with length of three, then s_1 includes the alphanumeric representation of the house number.

Each si has an associated time region (τ_i), the temporal element in which the IS (if $i = 0$) or the schema generated by applying changes $ic_1, \ldots, ic_i$ on IS, is applicable. Given n schema changes $ic_1, \ldots, ic_n$, τ_i ($0 \leq i \leq n$) is calculated as follows:

$$\tau_i = \begin{cases} t_v(IS) & \text{if i=n=0} \\ t_v(ic_i) - \bigcup_{j=i+1}^{n} t_v(ic_j) & \text{if } i < n \wedge n > 0 \\ t_v(ic_n) & \text{if } i = n \wedge n > 0 \end{cases}$$

The time region (τ_i) of a schema si is the part of the t_v of ic_i that was not overridden by later changes.

Let TAG^{s_i} be a temporal agent. TAG^{s_i} is an application program that executes the update operations that are applicable in its time region (τ_i). TAG^{s_i} is associated with the schema si.

Let ic_n be the n-th incremental change of the initial schema, and $t_v(ic_n)$ is its valid time. The following operations are required when a modification in the schema level occurs.

1. IC:=append(IC, ic_n). The n-th incremental change of the initial schema is added to IC.
2. A new schema s_n is generated by applying ic_n on s_{n-1}.
3. The ATD is recomputed to be $ATD := old(ATD) \cup t_v(ic_n)$, where $old(ATD)$ is the ATD that was valid prior to ic_n.

4. A new temporal agent TAG^{s_n} is added to the application with $\tau_n = t_v(ic_n)$.

5. For each temporal agent TAG^{s_i} $(0 \leq i \leq n-1)$, $\tau_i = old(\tau_i) - \tau_n$, where $old(\tau_i)$ represents the time region of the temporal agent TAG^{s_i} prior to the introduction of the incremental change ic_n.

For example, an application time domain of [Jan 1 1990, ∞) is the initial time region of a temporal agent TAG^{s_0}. The change ic_1 modifies the domain of the property House-Number to be valid in the interval [Jan 1 1991, ∞). A new schema s_1 is generated along with a new temporal agent TAG^{s_1} and a time region of [Jan 1 1991, ∞). The time region of TAG^{s_0} is now reduced to be the time interval [Jan 1 1990, Jan 1 1991).

Lemma 1 Partition of the application time domain

The time regions of all the temporal agents, present at a given time point t*, are a partition of the application time domain.*

The proof to Lemma 1 is presented in Appendix A. It entails proving the following two assertions:

Ass1: $\bigcup_{i=0}^{n} \tau_i = ATD$

Ass2: $\forall i, j \ (0 \leq i < j \leq n) \ \tau_i \cap \tau_j = \emptyset$

The proof is done using induction on the number of agents.

3.2 The single agent update process

This section briefly presents the update process of a single agent under a single schema. An elaborated discussion of the single agent update process is available in [12].

3.2.1 Basic assumptions

Temporal update languages have traditionally provided just the most basic update capabilities, that is, insertion, modification and deletion of variables. Due to the temporal properties, the update language gains new capabilities that have not been discussed in previous studies. Building new primitives to support these update capabilities would make a proper use of the potential power of temporal databases. The update process is based on the following three assumptions.

A1: The database is an **append only** database. New information is added, while existing information is left intact. The append only approach is necessary to support operations that require past database states. Unlike some other models that assume an append-only database (e.g., [1]), we use this assumption in the strictest fashion, i.e., no changes can be made to a state-element after the transaction that generated it has committed.

Due to the append only property of the temporal database, all the update operations add new state-elements to the database; there are no physical deletions,[1] or modifications of existing data.

A2: No a priori assumption about the selection of valid state-elements is made during the update process. For example, the update process allows the insertion of several state-elements of a variable with different values and intersecting t_v's. An immediate result of this assumption is that all the existing state-elements in the database remain accessible after any update operation.

A3: The update process allows the omission of any reference to temporal characteristics of the updated values. Defaults that are compatible with conventional databases' assumptions are enforced if the temporal characteristics are omitted.

3.2.2 Update operation types

The update process consists of transactions in a similar way to conventional databases. A transaction consists of retrieval and update operations. The following eight update operations are supported: insert, modify, suspend, disable, resume, freeze, unfreeze, and revise.

Insert This operation inserts a new object into the database. Along with the object insertion, the user may assign initial values to some or all the object variables. For example, a new person's data is presented to the database. The database generates a new instance of the class Person and initializes its values.

Modify This operation adds new information about an existing object. For example, on November 11 1991 the zipcode of John Doe has been changed to "12445" as of January 1 1992. The database modifies the Zipcode variable, keeping the old values as well.

As argued by some researchers (e.g., [7]), the modification of an object value in a temporal database changes the value in a given interval while keeping the earlier historical values of the object. This is different from modification of an object value in non-temporal databases in which the new value

[1] Storage limitations may require physical deletions in which case a scheme such as [16] may be used.

replaces the previous one. A modify operation can be applied to a time point different from the present, to an interval or even to the entire database time-line. For example, the government decided to change the legal currency in the state, due to high inflation during the last years. The new currency is defined as: *1 new unit* $\equiv$ *10 old units*. Conversion of all the history to the new currency values is required when a single scale should be used. Consequently, each value (historical and current) of the variables related with the currency is divided by 10. It should be noted that a query issued with respect to a time point that is earlier than the currency change, returns values that are given in the old currency.[2]

Suspend This operation establishes a reversible constraint that disables access to an object during a certain valid time. We refrain from naming this operation "delete," since there is no physical deletion in append only databases. Corresponding with assumption **A1**, the suspend operation generates a new state-element of a system-defined variable Object-Status with the value "suspended."

Resume This operation activates a suspended object. As in the insert operation, the resume operation may be used to set the values of some of the object's variables.

Disable An operation that establishes an irreversible constraint to prevent access to the specified object, as of a certain valid time point. The object is considered to be non existing as of that time point. Unlike the suspend operation, the disable operation is final in the sense that disabled objects cannot be resumed, i.e., there is no reverse operation for disable.

Freeze This operation establishes a reversible constraint that prevents the modification of a variable, except in the case of revising erroneous values as explained below. For example, the social security number is given once in a life time and cannot be altered.

Unfreeze Any frozen data may be unfrozen. An unfreeze operation applied to a variable, designates the removal of the freezing constraint. Any modification to that variable is allowed from that time on.

Revise This operation "corrects" an erroneous value of a variable at certain time points. It tags values that currently exist in the database as false ones and adds a new correct value instead. The revise operation allows the replacement of a frozen value, marking the previous value as an erroneous one.

The combination of revise operation and modify operation, makes a semantic distinction between a change in the real world and a correction of a wrong value that was reported to the database. The user can instruct the database to include or exclude the revised values in retrieval operations.

[2]It is possible to modify a variable, using its older values. This is done using *rules*, as discussed in [9].

Analyze-Operation:
input: operation, object-identifier, property-name, value, t_v, t_d
1 if $t_v \not\subset \tau_i$ then
2 begin
3 SE=$(se_1, \ldots, se_n)$:= *Retrieve* (Property-Name, $[t_e(\tau_i),\ t_e(t_v))$)
4 SE′:=$se \mid se \in$ SE $\wedge$ se.value= property-name
5 $t_{SE'} := \bigcup_{se \mid se \in SE'} \text{se}.t_v$
6 if $t_{SE'} \neq \emptyset$ then
7 begin
8 t_{up}:= $t_v - t_{SE'}$
9 temp:=$t_v - t_{up}$
10 ag:=receive-agent(temp)
11 Send-Update-Message (ag, operation, object-identifier, property-name, value, temp, t_d)
12 else t_{up}:= t_v
13 end
14 else t_{up}:=t_v
15 end
16 Execute-Update-Operation (operation, variable-name, value, t_{up}, t_d)

Figure 4: The agent Analyze-Operation algorithm

3.3 The parallel inter-agent update process

Each update transaction contains a sequence of update operations that are initiated by the user. Each update operation is decomposed by the transaction manager into primitive operations, such that each primitive operation is designated to update a single variable. Each primitive operation is a sequence ⟨operation-type, variable-name, value, t_v, t_d⟩, where the last three components are the values used in the generation of new state-elements. The update operations are sent to the execution manager.

Let $t_v = [t_s, t_e)$ be the valid time of a primitive operation. The execution manager assigns the primitive operation to a temporal agent TAG^{s_i} if $t_s \in \tau_i$. This policy assigns each primitive operation to the temporal agent whose time region includes the starting point of the operation's t_v. The assignment is unique, since by Lemma 1, the application time domain is fully partitioned by the time regions.

Each temporal agent TAG^{s_i} executes independently the primitive operations assigned to it. Each primitive operation is analyzed and executed. The analysis of each agent is intended to minimize both redundant operations and redundant state-elements. The agent's analysis algorithm is presented in Figure 4, and explained below.

An agent receives a primitive operation. At this point the following two scenarios are possible.

1. The valid time argument is totally included in the agent's time region. In this case the t_{up}, the actual

valid time, is set to the original t_v (line 14).

2. Some of the valid time is not included in the agent's time region. This scenario is handled in lines 2-13. The agent uses the function *Retrieve* to receive all the variables that were affected by the schema evolution in the interval starting at the end of its time region and ending at the end of the t_v of the operation. At this point there are two possible situations, as follows.

   (a) The agent is capable of processing the entire valid time of the operation if there was no change in the schema that is relevant to the property given as an argument to the *Analyze-Operation* routine. In this case, the t_{up} is set to the original t_v (line 12). This situation provides a way of minimizing both the agents activities and the storage area. All the agents with time regions intersecting t_v should act according to the same decision rule, since there are no schema changes relevant to the updated property in t_v. Consequently, there is no need to divide the update operation among several agents, and to divide the state-element into several state-elements, thereby saving storage area.

   (b) A temporal agent TAG^{s_i} cannot handle alone an update with a temporal element that is out of the time region of TAG^{s_i}, when a modification of the schema that affected the property is involved in the update operation. In this case, TAG^{s_i} selects a temporal agent ag that is capable of assisting it in executing the update operation, based on the knowledge stored in the class Meta-Data-Changes. TAG^{s_i} sends to ag a primitive operation with the relevant data, including the part of the temporal element of the update that is out of TAG^{s_i}'s control.

   In the *Analyze-Operation* routine, if the property was involved in one of the schema changes, then the agent excludes from t_v the time prior to the change (line 9), searches for the agent in charge of the remaining time (line 10) and sends an update message with the appropriate arguments (line 11).

Line 16 executes the update operation, using the t_{up} as the actual valid time assigned to this operation. The update process procedure is handled by each temporal agent independently.

Figure 5 is an example of a decision of TAG^{s_0} about an update operation which is not entirely included in its time region. During the period [Jan 1 1990, ∞) two schema changes occurred. The first change is of the domain of the House-Number, and is valid as of Jan 1 1991. The second change is of the domain of the Zipcode and is valid as of Sep 1 1991. A primitive update operation that updates the zipcode of John Doe to be "1w2zk" has a validity interval of [Sep 1 1990, ∞). According to the *Analyze-Operation* routine, TAG^{s_0}

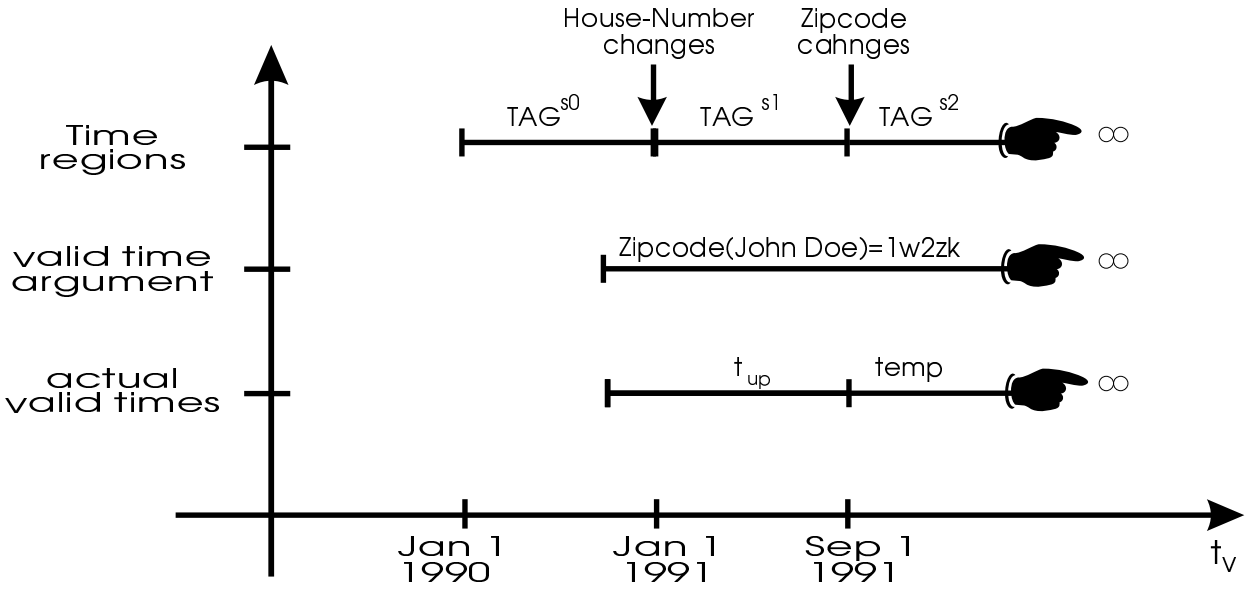

Figure 5: Graphical presentation of a scenario

receives the primitive update operation and calculates its t_{up} to be [Sep 1 1990, Sep 1 1991). A message is sent to TAG^{s_2} to update the same variable in temp=[Sep 1 1991, ∞).

3.4 Consistency modes

The execution manager decomposes an update operation to a set of parallel transactions. An important design issue is to determine the mutual consistency requirements among the distinct transactions. A similar problem has been examined in the context of derived data [8] resulting in the definition of the following mutual *consistency modes* of subtransactions.

fully consistent: A subtransaction *st1* is *fully consistent* with respect to a subtransaction *st0*, if their execution is atomic and both of them are executed within a single transaction.

loosely consistent: A subtransaction *st1* is *loosely consistent* with respect to a subtransaction *st0*, if the subtransactions can operate in two separated transactions with no mutual dependencies.

quasi consistent: A subtransaction *st1* is *quasi consistent* with respect to a subtransaction *st0*, if they can operate in two separate transactions. However, if one fails, the other rollbacks by issuing a compensating transaction.

In our context, the consistency modes are applicable in the inter-agent level, referring to the desired consistency requirements among various temporal agents, i.e., what is the consistency requirement among data that is valid in different time regions. These decisions can be made by the system designer in the process of the schema evolution.

In the **fully consistent** mode, the subtransactions can still run in parallel, simulating a single transaction by having commit/abort dependencies. For example, a subtransaction *st0* cannot commit before the subtransaction *st1* commits, and a subtransaction *st1* should abort if the subtransaction *st0* aborts.

4 Simulation results

We have implemented the algorithm presented in Section 3.3 to evaluate the feasibility of the temporal agents approach. In our experiments, we have concentrated on the following issues:

1. A cost-benefit analysis of using temporal agents as a function of the transaction length.
2. The impact of different assumptions for the inter-agent communication overhead on the cost-benefit analysis described above.
3. The impact of consistency modes on the success and failure rates of transactions in this environment.

The program was written in C for a single processor, simulating a parallel computer. A version of the simulation program for a parallel computer is now under development. We performed the tests on a Sun 690 running SunOS 4.1.3, and the tests were compiled using the **GNU C Compiler (gcc)**.

4.1 A cost-benefit analysis of using temporal agents

In this simulation, we compared the following three possible strategies for handling schema versioning.

1. The database uses a single "active" schema at all valid time points. Using this method, the active schema is normally the schema that is the result of the last schema modification. This method supports schema evolution, by allowing changes in the schema level, but does not support schema versioning [15].

2. The database supports schema versioning by handling each schema version separately, in a sequential manner. This method uses a simple update algorithm, with no need for communication protocols.

3. The database supports a multi-agent model, as presented in this paper. The parallel processing in the multi-agent model have performance cost and performance benefits. On the one hand, using parallel computing decreases the computation time. On the other hand, the synchronization, coordination and communication among agents generate a performance overhead.

The simulation evaluated the total performance of a transaction as a function of the number of operations in a transaction, and the method of applying schema versioning. At each run of the simulation, the communication overhead was assumed to be a constant. Figure 6 presents an instance of this simulation set, with an overhead of 12%. A set of 100 transactions, each one consists of a similar number of operations, was tested on the three strategies of schema versioning support. One with no schema versioning support, the other with a sequential support of two schema versions that halved the application time domain, and the last with a multi-agent model that supports two schema versions that halved the application time domain. In all cases, the application time domain was an interval of the form [0, ν), where ν=300. Each update operation updated a variable with a valid time $[t_s, t_e]$, such that $t_s \sim$ U[0, ν] and $t_e \sim$ U[t_s, ν]. The update operations were designed to succeed in all of the time regions; thus, no failure overhead should be considered, and the only trade-off in a multi-agent model is the cost of communication vs. the benefit of parallel execution.

According to the results presented in Figure 6, the use of sequential handling of multi-schemata has the worst performance of all three strategies. The interesting result of this simulation is that when the number of operations in a transaction is 4 or above, and the communication overhead is 12%, it is more beneficial to support schema versioning with a multi-agent model than not supporting schema versioning at all.

4.2 The impact of different assumptions for the inter-agent communication overhead

As stated above, the major cost of parallel processing is the overhead generated by the need to coordinate and to transfer information among agents. The communication overhead in Figure 6 was assumed to be a constant (12%). Figure 7 presents the result of several simulations that evaluated the sensitivity of the update process to changes in the communication overhead, the number of agents, and the number of operations in a transaction. The simulation sets assumed up to three agents in parallel. We limit the simulation parameters to a reasonable amount of operations in a transaction (up to 5 operations), and a reasonable communication

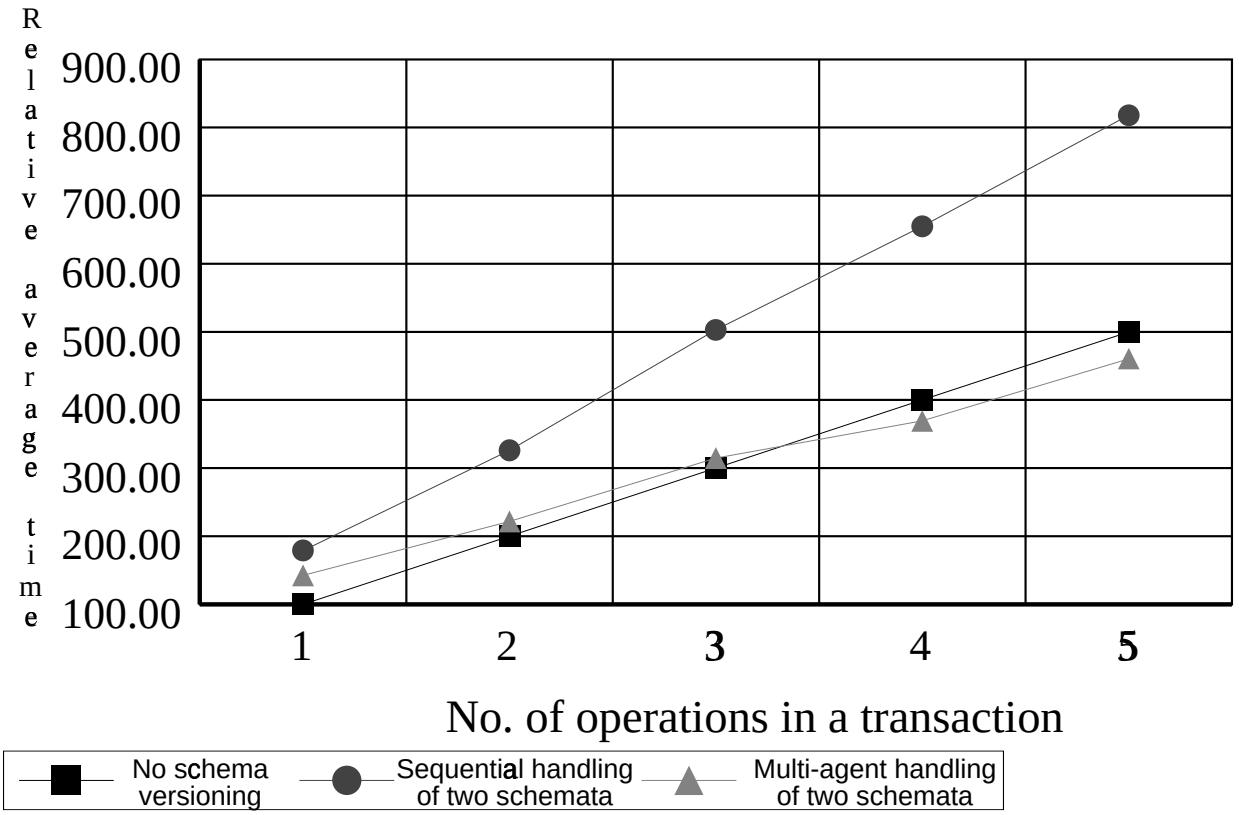

Figure 6: The average elapsed time of transactions for the three schema versioning handling methods

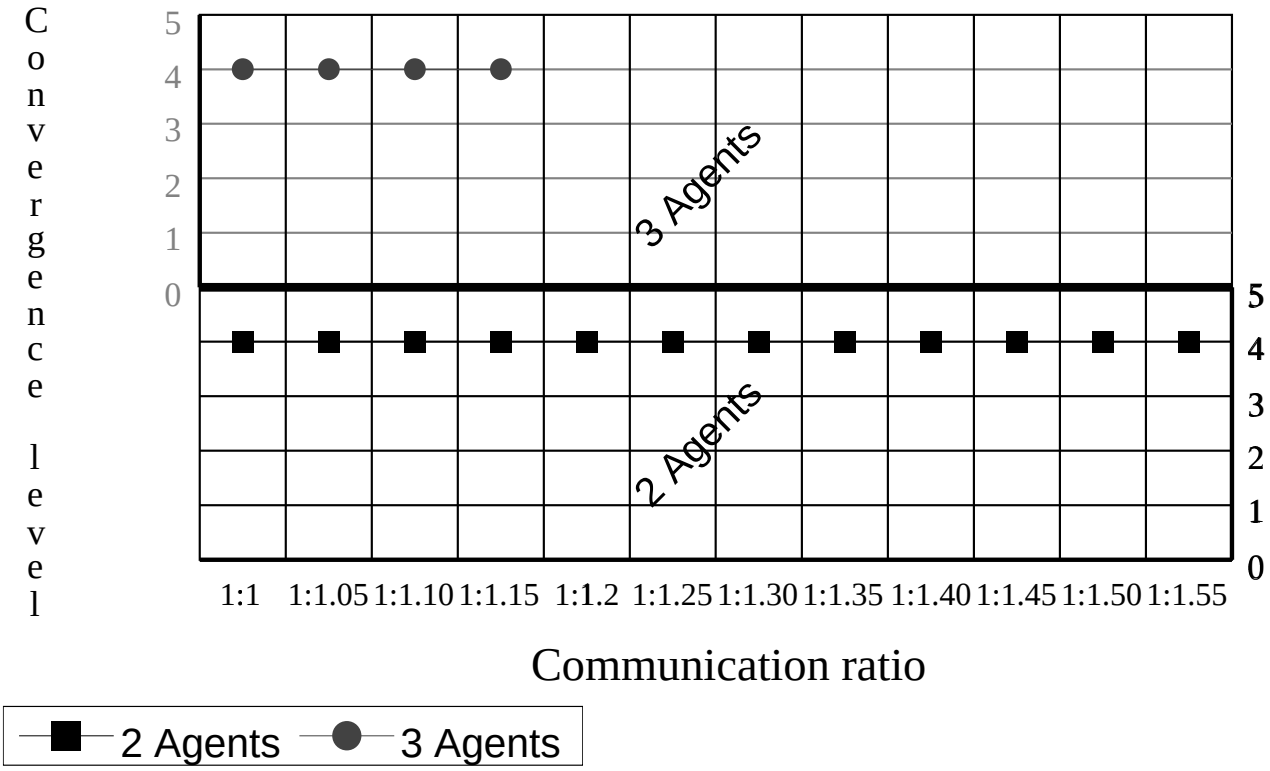

Figure 7: The break-even point of using the multi-agent model as a function of communication overhead

overhead (up to 60%).

Figure 7 presents the break-even point of the cost-benefit analysis in using the multi-agent model vs. the use of a single schema with no schema versioning support. It contains two parts that denote separate simulation sets for the two agents case, and the three agents case. In each part, we show the minimal number of operations per transaction for which it is beneficial to use the multi-agent model with the given amount of agents. For two agents, the simulations show that the break-even point is achieved when the number of operations per transactions is 4, regardless of the communication overhead. The range of communication overheads that was used in the simulations covers all reasonable values (up to 55%). For three agents, the simulations show that the break-even point is achieved when the number of operations per transactions is 4. This result is valid for a communication overhead of up to 15%. When the communication overhead is higher than that, the break-even point was above five operations per transaction. Thus, these results are out of the tested range.

For two agents, it is beneficial to use the multi-agent model regardless of the communication overhead, for

transactions with 4 operations or more. For three agents, the multi-agent model is preferred for transactions with 4 operations or more, and for a communication overhead of up to 15%. It is worth noting, that the alternative method is not to support schema versioning at all. Thus, these results show the benefits of using the multi-agent model due to time considerations only, while increasing the capabilities of the supporting database.

4.3 The impact of consistency modes on the success and failure rates

There are several possible consistency modes among sub-operations that were constructed from the same operation and that relate to different time regions. A *fully consistent* mode requires the success of each of the sub-operations as a condition for the success of the operation itself. A *loosely consistent* mode enables partial success of the operation, i.e., success in only part of the time regions.

In this set of simulations we check the sensitivity of the transaction's success rate to the consistency modes and the number of operations in a transaction. In the *fully consistent* mode, a transaction can either succeed or fail. In the *loosely consistent* mode, each operation can be partitions into sub-operations, each handled by a different agent. Each sub-operation may succeed or fail independently. A transaction *succeeds* for a given agent iff all of the sub-operations associated with an agent have succeeded, and *fails* otherwise. A transaction rate of success is:

$$\frac{\text{The number of agents for which the transaction succeeds}}{\text{The number of agents that participated in the transaction}}$$

A set of 100 transactions, each one consists of a similar number of operations, was tested on a temporal database with three schemata that equally partitioned the application time domain. The application time domain was an interval of the form $[0, \nu)$, where ν=300. Each update operation updated a variable with a valid time $[t_s, t_e]$, such that $t_s \sim \text{U}[0, \nu]$ and $t_e \sim \text{U}[t_s, \nu]$. The three schemata accept as valid values the values $\{0 \ldots 99\}$ (first schema), $\{0 \ldots 999\}$ (second schema), and $\{0 \ldots 9999\}$ (third schema). The update operation uses a value $val \sim \text{U}[0, 120]$ if t_s is in the first schema, a value $val \sim \text{U}[0, 1200]$ if t_s is in the second schema, and a value $val \sim \text{U}[0, 12000]$ if t_s is in the third schema. Thus, the update operations were designed to produce approximately 20% failure in their starting agent's time region. It is worth noting that we deliberately inflated the failure rate, to make sharp distinctions between the different consistency modes.

Figure 8 presents the simulation results of transactions with a *fully consistent* protocol. The transaction's success rate rapidly decreases as the number of operations in a transaction increases. The success rate of

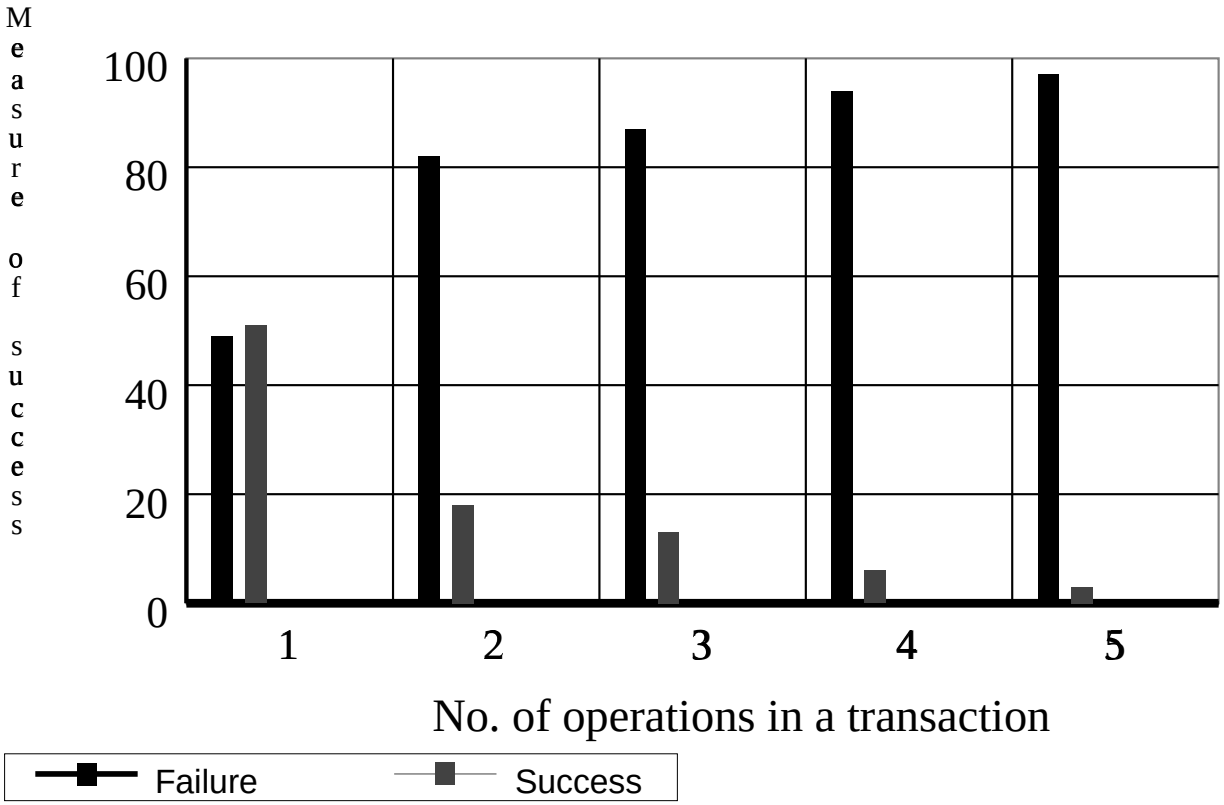

Figure 8: Transactions success rate for a *fully consistent* protocol

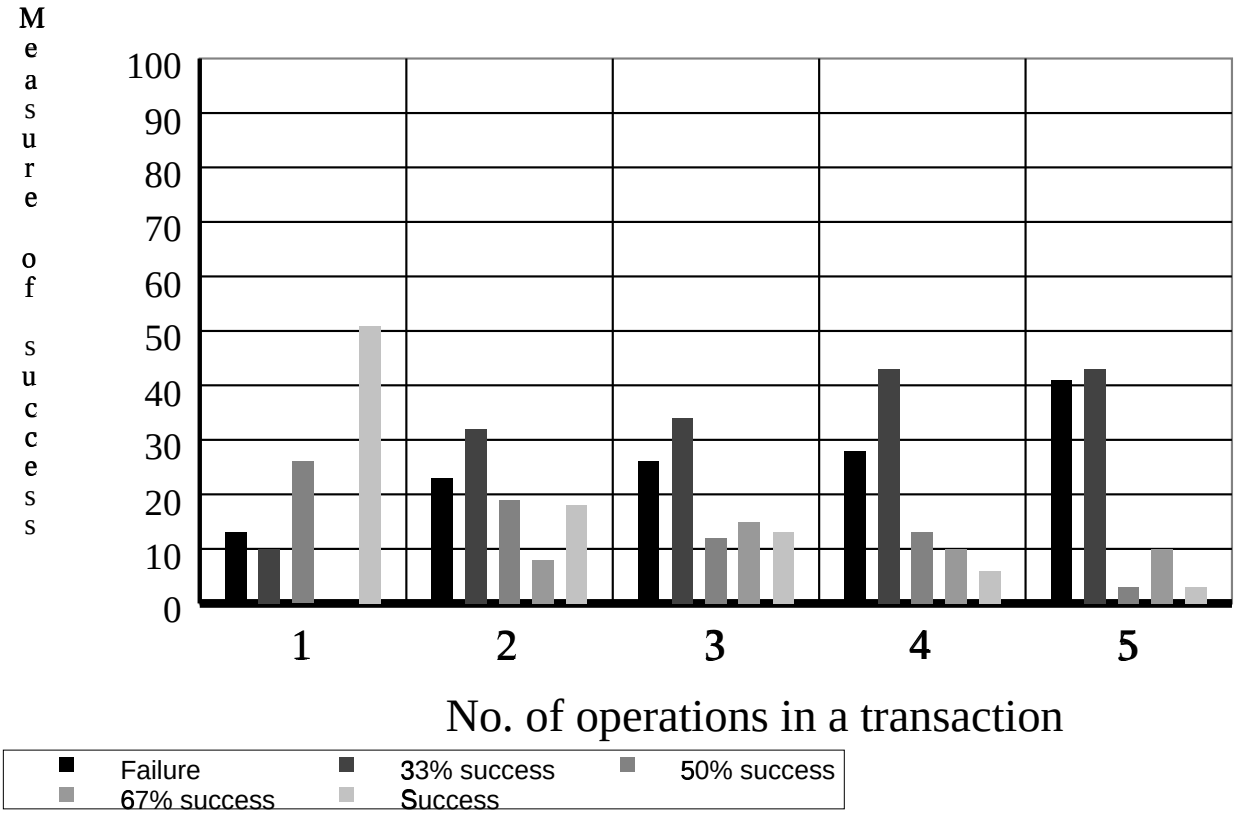

Figure 9: Transactions success rate for a *loosely consistent* protocol

transactions with five operations is very low (3%). Figure 9 presents the simulation results of transactions with a *loosely consistent* protocol. The success rate is significantly higher than the *fully consistent* case. The number of failures in a transaction with five operations is about 40%, and a similar number of transactions has a 33% of success.

The experiments show that it is beneficial to use the *loosely consistent* mode in a temporal database that supports schema versioning. Thus, the autonomy of the commit decision is delegated to the agent level rather than the transaction level. It is worth noting that researchers in the transaction process research area have recognized the need to compromise the transaction atomicity for certain types of applications [14].

5 Related work

In this section we present works in related research areas, namely, parallel processing, schema evolution and updates in temporal databases.

5.1 Parallelism in databases

Parallel processing technologies are used in the database systems area as a general solution of the *I/O bottleneck* [3], influenced by high disk access time with respect to main memory access time. The proposed general solution is "increasing the I/O bandwidth through parallelism" [30]. This hardware solution should be accompanied by software solution in areas such as operating system support, data placement, parallel database programming languages, and transaction management. Many problems in these areas are still open [30].

There are three inherent parallelism types in data processing [30]. *Interquery parallelism* enables the parallel execution of multiple queries generated by concurrent transactions. *Intraquery parallelism* allows a parallel execution of operations within the same query, and *intraoperation parallelism* permits an operation to be partitioned to suboperations using *function partitioning*. An example of the latter type is the parallel join operation [5].

The subject of parallel query processing has been researched by [4], [17], [26] and others in the areas of relational databases and deductive databases.

The work presented in this paper is aimed at providing an appropriate mechanism for updating temporal databases by taking advantage of parallel database systems and enabling intraoperation parallelism. We enable parallel update processing of temporal databases by using function partitioning based on time, thus using the unique properties of this database research area.

5.2 Schema evolution

Over the last years, several papers discussing different aspects of schema evolution have been published. A thorough bibliography [23] contains approximately 40 papers regarding schema evolution in the relational data model, the object-oriented paradigm and others.

Several papers (e.g., [24]) revised basic query languages to accommodate the effect of schema evolution while others (e.g., [20]) considered the update algebra required for handling evolving schema. The latter, however, failed to use the valid time along with the transaction time, and by that disallowing retroactive and proactive updates.

Data conversion, as a result of a change in the schema is discussed in several papers, including [22], [28]

and [2]. None of these works approached the update complexity problem.

5.3 Updating temporal databases

Most of the related work of updating temporal database remained within the conventional database operation types of *insert, modify* or *delete* while giving these operations slightly different meaning. For example, according to some researchers (e.g., [7]) the difference between updates in temporal databases and non-temporal databases is that modification of an attribute value will make a temporal change while keeping the earlier historical values of the attribute rather than actually replacing the previous attribute value. Others (e.g., [19]) expanded the *modify* operation to include meta-data, thus allowing schema evolution in time, as well as data evolution.

Some studies ([18]; [27]; [1]; [25]) distinguished a *modification* from a *correction*. *Modifying* a data is carried out when the situation in the real world has changed. *Correcting* a data is carried out because an error in the existing data has been detected.

In [13] we provide an extended set of update operation types that consist of: insert, modify, suspend, disable, resume, freeze, unfreeze and revise. These operation types are described in Section 3.2.

6 Conclusion

This paper presents a description of a parallel execution model for temporal databases, to overcome the performance problems of a temporal database. Decision support systems, as a major example, require fast response time in many cases, while the functionality requirements are complex and include support of retroactive and proactive updates coupled with evolving schema. This model has the following properties:

- The schema history is represented as a collection of temporal regions that issue a natural partition of the application time domain. Each temporal region is processed by a temporal agent. The concept of temporal agents enables parallel execution of updates in different time regions. The collaboration among agents allows different levels of dependencies among subtransactions.
- The model supports retroactive and proactive updates. The use of t_v to denote the validity of any component of the information system gives the flexibility to modify past versions that may still be useful for some users, generate new versions that refer to the past and generate and modify future

versions.

- The model supports schema evolution and a complete documentation about the past versions of the schema. Different versions of the schema are handled concurrently using high level abstractions. It is possible to change variables that are related to several versions of the schema in one operation. This property is especially important in cooperative systems in which different users possess different versions of the schema. The alternative, employed in current systems, is to perform manually all the implications upon the distinct versions.

An ongoing research concentrates on the combination of an active dimension in a temporal database. The active dimension generates interdependencies among different temporal agents, thus the update process should be modified to ensure a correct and efficient transaction execution in this framework.

A prototype of the system is currently being developed. Further research will deal with various issues, including exception-handling in the proposed environment, hypothetical change management to support possible worlds and the planning of schema evolution.

A further research will also consider the problem of retrieving a temporal active database with evolving schema, where unlike the update process, several different schemata are available at the same time point, but only one of them should be selected, based on the query observation point.

Acknowledgments

The basic temporal model has been constructed in collaboration with Arie Segev. Miriam Kaspi-Flor wrote the simulation program, based on the algorithm for temporal agents in a temporal database. We thank Dov Dori for his useful remarks.

Appendix A: The proof of Lemma 1

Lemma 1 *The time regions of all the temporal agents, present at a given time t, is a partition of the application time domain.*

Proof:
The proof of Lemma 1 entails proving the following two assertions:

Ass1: $\bigcup_{i=0}^{n} \tau_i = ATD$

Ass2: $\forall i, j \ (0 \leq i < j \leq n) \ \tau_i \cap \tau_j = \emptyset$

1. Let DB be a database of an application with an application time domain ATD.

   Let $TAG^{s_0}, \ldots, TAG^{s_{(n-1)}}$ be n temporal agents of DB with corresponding time regions $\tau_0, \ldots, \tau_{n-1}$.

   The proof is done using induction on the number of agents.

2. **[Induction Base:]** n=0

   $\Longrightarrow \bigcup_{i=0}^{n} \tau_i = \tau_o$.

   By definition, $\tau_o = t_v(IS) = ATD$ when n=0.

   $\Longrightarrow \bigcup_{i=0}^{n} \tau_i = \tau_0 = ATD$ **(1)**.

   Trivially, $\forall i, j \ (0 \leq i < j \leq n) \ \tau_i \cap \tau_j = \emptyset$ **(2)** $\square$

3. **[Induction Step:]** Assume $\tau_0, \ldots, \tau_{n-1}$ is a partition on ATD.

   (a) Given n temporal agents of DB, the effects of a schema change results in:

      i. A new temporal agent TAG^{s_n} is added.

      ii. $ATD = old(ATD) \cup \tau_n$ **(3)**

      iii. $\tau_n = t_v(ic_n) = \{t \mid t \in t_v(ic_n)\}$. **(4)**

      iv. For each temporal agent TAG^{s_i} $(0 \leq i \leq (n-1))$:

      $\tau_i = old(\tau_i) - \tau_n = \{t \mid t \in old(\tau_i) \wedge t \notin \tau_n\}$ **(5)**

      (3) $\Longrightarrow old(ATD) \subseteq ATD$ **(6)**

      (3) $\Longrightarrow \tau_n \subseteq ATD$ **(7)**

      (5) $\Longrightarrow \tau_i \subseteq old(\tau_i)$ **(8)**

      (4), (5) $\Longrightarrow \tau_i = \{t \mid t \in old(\tau_i) \wedge t \notin t_v(ic_n)\}$ **(9)**

      the induction assumption $\Longrightarrow \bigcup_{i=0}^{n-1} old(\tau_i) = old(ATD)$ **(10)**

      (6), (10) $\Longrightarrow old(\tau_i) \subseteq ATD$ **(11)**

      (8), (11) $\Longrightarrow \tau_i \subseteq ATD$ **(12)**

      (7), (12) $\bigcup_{i=0}^{n} \tau_i \subseteq ATD$ **(13)**

   (b) **(5)** $\Longrightarrow \bigcup_{i=0}^{n} \tau_i = \{t \mid t \in old(\tau_1) \wedge t \notin t_v(ic_n)\} \cup \ldots \cup \{t \mid t \in old(\tau_{n-1}) \wedge t \notin t_v(ic_n)\} \cup \tau_n$. **(14)**

   (4), (14) $\Longrightarrow \bigcup_{i=0}^{n} \tau_i = \{t \mid t \in old(\tau_1) \wedge t \notin t_v(ic_n)\} \cup \ldots \cup \{t \mid t \in old(\tau_{n-1}) \wedge t \notin t_v(ic_n)\} \cup t_v(ic_n)$

(15)

(15), De-Morgan rules $\Longrightarrow \bigcup_{i=0}^{n} \tau_i = \{t \mid (t \in old(\tau_1) \vee \ldots \vee t \in old(\tau_{n-1})) \wedge t \notin t_v(ic_n)\} \cup t_v(ic_n) =$
$\bigcup_{i=0}^{n-1} old(\tau_i) - t_v(ic_n) \cup t_v(ic_n) = old(ATD) - t_v(ic_n) \cup t_v(ic_n)$ **(16)**

(3), (7), (16) $\Longrightarrow \bigcup_{i=0}^{n} \tau_i \supseteq ATD - t_v(ic_n) - t_v(ic_n) \cup t_v(ic_n) = ATD$ **(17)**

(13), (17) $\Longrightarrow \bigcup_{i=0}^{n} \tau_i = ATD$ **(18)**

(c) Given i, j such that $0 \leq i < j < n$:

(9) $\Longrightarrow \tau_i \cap \tau_j = \{t \mid t \in old(\tau_i) \wedge t \notin t_v(ic_n)\} \cap \{t \mid t \in old(\tau_j) \wedge t \notin t_v(ic_n)\} =$

$\{t \mid t \in old(\tau_i) \wedge t \in old(\tau_j) \wedge t \notin t_v(ic_n)\} =$

$old(\tau_i) \cap old(\tau_j) - t_v(ic_n)$ **(19)**

the induction assumption, (19) $\Longrightarrow \tau_i \cap \tau_j = \emptyset - t_v(ic_n) = \emptyset$ **(20)**

(d) Given i such that $0 \leq i < n$:

(4) $\Longrightarrow \tau_i \cap \tau_n = \tau_i \cap t_v(ic_n)$ **(21)**

(4), (5), (21) $\Longrightarrow \tau_i \cap \tau_n = \{t \mid t \in old(\tau_i) \wedge t \notin t_v(ic_n)\} \cap t_v(ic_n) =$

$\{t \mid t \in old(\tau_i) \wedge t \notin t_v(ic_n) \wedge t \in t_v(ic_n)\} = \emptyset$ **(22)**

(e) **(20), (22)** $\Longrightarrow \forall i, j \ (0 \leq i < j \leq n) \ \tau_i \cap \tau_j = \emptyset$ **(23)**

4. **(1), (2), (18), (23)** $\Longrightarrow$ The time regions of all the temporal agents, present at a given time t is a partition of the application time domain. □

References

[1] T. Abbod, K. Brown, and H. Noble. Providing time-related constraints for conventional database systems. In *Proceedings of the 13th International Conference on VLDB*, pages 167–175, Brighton, 1987.

[2] J. Banerjee, H.T. Chou, H.J. Kim, and H.F. Korth. Semantics and implementation of schema evolution in object-oriented database. *SIGMOD Record*, 16(3):311–322, 1987. ACM SIGMOD conference.

[3] H. Boral and D.J. DeWitt. Database machines: an idea whose time has passed? a critique of the future of database machines. In *International workshop on Database Machines, Munich*, 1983.

[4] U.S. Chakravarthy and J. Minker. Processing multiple queries in database systems. *Database Engineering*, 5(3):38–44, Sep 1982.

[5] D.J. DeWitt and R. Gerber. Multiprocessor join algorithms. In *International Conference of VLDB*, Stockholm, 1985.

[6] D.J. Dewitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[7] R. Elmasri and G. Wuu. A temporal model and query language for ER database. In *Proceedings of the International Conference on Data Engineering*, pages 76–83, Feb 1990.

[8] O. Etzion. Flexible consistency modes for active database applications. *Information Systems Journal*, 18(6):391–404, Nov 1993.

[9] O. Etzion, A. Gal, and A. Segev. Temporal active databases. In *Proceedings of the International Workshop on an Infrastructure for Temporal Database*, June 1993.

[10] O. Etzion, A. Gal, and A. Segev. Retroactive and proactive processing. In *Proceedings of Research Issues in Data Engineering - Active Database Systems*, pages 126–131, Feb 1994.

[11] S.K. Gadia. The role of temporal elements in temporal databases. *Data Engineering Bulletin*, 7:197–203, 1988.

[12] A. Gal. *TALE — A Temporal Active Language and Execution Model*. PhD thesis, Technion—Israel Institute of Technology, Technion City, Haifa, Israel, May 1995.

[13] A. Gal, O. Etzion, and A. Segev. Extended update functionality in temporal databases. Technical Report ISE-TR-94-1, Technion—Israel Institute of Technology, Sept. 1994.

[14] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 249–259, May 1987.

[15] C.S. Jensen et al. A consensus glossary of temporal database concepts. *ACM SIGMOD Record*, 23(1):52–63, 1994.

[16] C.S. Jensen and L. Mark. A framework for vacuuming temporal databases. Technical Report CS-TR-2516, University of Maryland, Dept of CS, College Park, MD 20742, August 1990.

[17] W. Kim. Global optimization of relational queries: A first step. In W. Kim, D. Reiner, and D. Batroy, editors, *Query Processing in Database Systems*, pages 206–216. Springer-Verlag, NY, 1984.

[18] M.R. Klopprogge and P.C. Lockmann. Modeling information preserving databases; consequences of the concept of time. In *Proceedings of the International Conference of VLDB*, Florance, Italy, 1983.

[19] E. McKenzie. *An Algebraic Language for Query and Update of Temporal Databases*. PhD thesis, Computer Science Department, University of North Carolina in Chapel Hill, Sep 1988.

[20] L.E. McKenzie and R.T. Snodgrass. Schema evolution and the relational algebra. *Information Systems*, 15(2):207–232, 1990.

[21] J. Mylopoulos and E. Yu. Aligning information system strategy with business strategy: A technical perspective. A Keynote Address in the Int. Workshop on Next Generation Technologies and Systems, Haifa, Israel, June 1993.

[22] D.J. Penny and J. Stein. Class modifications in the gemstone object-oriented dbms. *SIGPLAN Not.*, 22(12):111–117, 1987. Proc OOPSLA '87.

[23] J.F. Roddick. Schema evolution in database systems—an annotated bibliography. *SIGMOD Record*, 21(4):35–40, Dec 1992.

[24] J.F. Roddick. Sql/se—a query language extension for databases supporting schema evolution. *SIGMOD Record*, 21(3):10–16, Sep 1992.

[25] E. Rose and A. Segev. Toodm-a temporal, object-oriented data model with temporal constraints. In *Proceedings of the International Conference on the Entity-Relationship Approach*, pages 205–229, San Mateo, California, 1991.

[26] T.K. Sellis. Multiple query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, Mar 1988.

[27] R. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19:35–42, Sep 1986.

[28] L. Tan and T. Katayama. Meta operations for type management in object-oriented databases—a lazy mechanism for schema evolution. In *Proceedings of the First International Conference on DOOD*, pages 241–258, Kyoto, Japan, 1989. North-Holland.

[29] A.U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases*. The Benjamin/Commings Publishing Company, Inc., Redwood City, CA., 1993.

[30] P. Valduriez. Parallel database systems: Open problems and new issues. *Distributed and Parallel Databases*, 1(2):137–165, 1993.

[31] G. Wiederhold. From data engineering to information engineering, Feb 1994. A keynote address in International Conference on Data Engineering ICDE '94.