On Transaction Management in Temporal Databases

Avigdor Gal*

Department of Computer Science
University of Toronto

Abstract. A transaction model provides a framework for concurrent processing of retrieval and update operations in a database. Considerable research effort has focused on various techniques and protocols to ensure the ACID properties of transactions in conventional databases. However, the adoption of these techniques and protocols to temporal databases is not trivial. In particular, a refined locking mechanism based on temporal characteristics can provide better concurrency among transactions in temporal databases than a conventional locking mechanism. Accordingly, this paper presents a set of modifications and fine tuning of traditional concepts in transaction management, to enable a better performance of temporal databases. We also suggest a scheme for implementing a transaction protocol for temporal databases on top of a relational database. The contribution of the paper is in identifying the unique properties of transaction management in temporal databases and the use of these properties to provide a refined locking mechanism to enhance the concurrency of such databases. In particular, we show that the classic 2PL mechanism cannot ensure serializability in temporal databases. Instead, we suggest an alternative method to ensure serializability and reduce redundant abort operations, which is based on a temporal serializability graph.
Keywords: temporal databases, transaction management

1 Introduction

A transaction model provides a framework for concurrent processing of retrieval and update operations in a database. A conventional transaction model ensures the following properties (ACID):

Atomicity: Either all the operations of a transaction are properly reflected in the database or none are.
Consistency: Execution of a transaction in isolation preserves the consistency of the database.
Isolation: Each transaction assumes that it is executed alone. Any intermediate transaction results are not available to other concurrently executed transactions.
Durability: The values changed by the transaction persists after the transaction was successfully completed.

Considerable research was dedicated to various techniques and protocols to ensure the ACID properties of transactions in conventional databases, e.g. the locking mechanism and the 2PL (Two Phase Locking) protocol, using serializability as a correctness criteria. However, adopting these techniques to temporal databases [27], i.e. databases that enable the accumulation of information over time and provide the capability to store different values of the same data element with different time characteristics, is not trivial. When adopting conventional techniques to accommodate the needs of temporal databases, a refined locking mechanism based on temporal characteristics should be designed, to provide better concurrency among transactions in temporal databases. Also, conventional protocols cannot efficiently support transactions in temporal databases. For example, as suggested in [21] and demonstrated in this paper, the classic 2PL mechanism cannot ensure serializability in temporal databases. Therefore, the use of either a strict 2PL or a serial transaction processing is required, when using conventional methods, to prevent a non-serializable transaction management in temporal databases.

* The work was conducted while the author was at the University of Toronto. He is currently at the MSIS Department, Rutgers University, 94 Rockafeller Road, Piscataway, NJ 08854-8054

This paper presents a set of modifications and fine tuning of traditional concepts in transaction management, which are required for a better performance of temporal databases. To exemplify these modifications, we provide a scheme for implementing a temporal transaction protocol on top of a relational database model. The approach of using add-on temporal facilities with an existing conventional database model is considered nowadays the most suitable approach to provide temporal capabilities in databases [28].

The contribution of the paper lies in identifying the unique properties of transaction management in temporal databases and the use of these properties to provide a refined locking mechanism to enhance transactions' concurrent execution in such databases. In particular, we provide an alternative method to 2PL, based on a temporal serializability graph, to ensure concurrency while reducing the number of redundant abort operations.

The issue of transaction modelling for temporal databases was suggested as one of the challenges for further research at the NSF International Workshop on an Infrastructure for Temporal Databases [4] and was first introduced in [21] and [30]. While the former relates to a transaction time temporal database only, the latter uses a simplified temporal data model and therefore results in a much simpler transaction model. In particular, the temporal database in [30] does not support transaction time and is not append-only. Some consideration to the issue of using commit time as a transaction time was given in [8], [19], and [28].

While several previous researches have discussed the refinement of transaction models (e.g. SAGAS [14] and ACTA [6]), none of them relate specifically to the unique properties of temporal databases. Nonetheless, it is worth noting that an extended model like SAGAS can serve as an underlying model for implementing better transaction models for temporal databases by using temporal independence and the refined locking mechanism presented in this paper.

Most transaction models deal with time by using histories and time stamps as useful tools for ensuring serializability, and some research was done on querying transaction logs to obtain temporal-oriented information [3]. Yet, these time considerations provide a different dimension than the one we handle in this paper, i.e. providing temporal databases with a coherent transaction model. Time stamping mechanisms for ensuring serializability were discussed in the framework of conventional databases [2] and some research was even dedicated to multiversion systems [16]. While this area of research bares similarity to the research presented in this paper, several major differences exist. First, the time stamping does not provide temporal capabilities on top of a conventional database. Second, a transaction in some temporal database types (e.g. bi-temporal databases) is time stamped at commit time, rather than at the beginning of its execution. Therefore, as we show in this paper, the assumptions that hold for a time stamping mechanism are not valid for transactions in bi-temporal databases.

The rest of the paper is organized as follows. Section 2 provides a data model and an execution model of a temporal database that is utilized throughout the paper. A transaction model for temporal databases is introduced in Section 3 followed by a scheme for implementing a temporal transaction protocol on top of a relational database model (section 4). Section 5 concludes the paper.

2 A data model for temporal databases

This section introduces the basic concepts of a data model for temporal databases. The terminology is based on [10], and it uses a semantic data model which is more adequate for representing sets of sets, a common requirements in temporal databases. The generic model can be easily translated into a relational as well as an object-based data model (see [10] for details). An *object* is defined as an instance of a class or a tuple in a relation and a *property* is defined as an attribute in the object-based model and a column in the relational model. The term *class* defines either a class in the object-based model, or a relation in the relational model. Let $DBS = \{C_1, C_2, ..., C_m\}$ be a database schema that consists of m classes. A class C_i has n_i properties $P_1^i, P_2^i, ..., P_{n_i}^i$, each with a domain $Dom(P_j^i)$, where a *domain* is a set of values. An instance of a property P_j^i is an element of the set $Dom(P_j^i)$, represented as $\alpha.P_j^i$, where α is an object identifier instance of the

appropriate class, a class name, or a variable. A *class domain* of a class C_i ($CDOM(C_i)$) is a subset of the Cartesian product $Dom(P_1^i) \times Dom(P_2^i) \times \ldots \times Dom(P_{n_i}^i)$. An *object state os* of an instance o of a class C_i at time t is an element $\langle p_1, p_2, ..., p_{n_i} \rangle \in CDOM(C_i)$. An *application state* at t is a set $\{os(o) \mid o$ is an instance of C_i $(1 \leq i \leq m)$ at $t\}$.

Following previous works in the temporal database area, we adopt a discrete model of time [7], isomorphic to the natural numbers. Hence, a *temporal domain* is a domain $T \cong \mathbf{N}$. The discrete model defines a *Chronon* [17] to be a nondecomposable unit of time ($t \in T$), whose granularity is application dependent. A *time interval* is designated as $[t_s, t_e)$, the set of all chronons t such that $t_s \leq t < t_e$. A *temporal element* [9] is a finite union of disjoint time intervals.

The temporal infrastructure document [26] advocates a bi-temporal database model, in which each data element is associated with two temporal dimensions, called valid time and transaction time. A *valid time* (v) is a temporal element that designates the collection of chronons at which the data element is considered to be true in the modeled reality. A *transaction time* (x) is a chronon that designates the time in which the transaction that inserted the data element's value to the database was committed. Therefore, in a bi-temporal database, a domain $Dom(P_j^i)$ of an attribute P_j^i is the Cartesian product of three domains, one of which is the value domain of the property, while the other two are temporal domains.

Information about an object is maintained as a set of *variables* (instances of the class' properties), where each variable contains information about the history of the values of the property. Each variable is represented using a set of *state-elements*, where a state-element se is an element of a domain that consists of a value ($se.value$) and temporal characteristics ($se.v$ and $se.x$ in the bi-temporal case). The following definition provides some properties of sets of state-elements:

Definition 1. Let SE_1 and SE_2 be two sets of state-elements of a variable $\alpha.P$:

- SE_1 and SE_2 are *identical* iff
  $\forall 1 \leq i, j \leq 2(\forall se \in SE_i \; \exists se^{'} \in SE_j \mid se.value = se^{'}.value \wedge se.v = se^{'}.v \wedge se.x = se^{'}.x)$.
- SE_1 and SE_2 are *similar* iff $\forall 1 \leq i, j \leq 2(\forall \{se_1^{'}, se_1^{''}\} \subseteq SE_i \; \exists \{se_2^{'}, se_2^{''}\} \subseteq SE_j \mid$
  1. $se_1^{'}.value = se_2^{'}.value \wedge se_1^{'}.v = se_2^{'}.v \wedge$
  2. $se_1^{''}.value = se_2^{''}.value \wedge se_1^{''}.v = se_2^{''}.v \wedge$
  3. $se_1^{'}.x \circ se_1^{''}.x \longrightarrow se_2^{'}.x \circ se_2^{''}.x \; (\circ \in \{<, =, >\}))$

Based on Definition 1, the similarity of two sets of state-elements identifies two sets that consist of the same information, and were committed at the same order, yet in different chronons.[2]

Various database models accumulate state-elements in different ways. Some models (e.g. TALE [13]) follow the *append only* approach, according to which new information is added while existing information is left intact. Other models (e.g. [1]) follow the *alternative* approach, according to which a new state-element of a variable in a valid time τ replaces any other state-element in τ. Many hybrids exist between these two extremes. If the data model supports the append-only approach, previous inserted state-elements can be accessed using an observation time abstraction (see below).

Temporal relationships for retrieval and update purposes are specified through the use of *participants*, variables with valid time binding, of the form $\langle \alpha.P, v \rangle$. A variable followed by a v defines the state-elements that are retrieved by the operation, or the bounded effect of the operation on the generation of new state-elements.

[2] In [18], value equivalence is suggested, where all temporal characteristics are being stripped off. This is, however, a different notion than *similarity*.

An append-only temporal database is updated by adding state-elements to variables, therefore generating a new application state each time a state-element is added. Each state-element is associated with two temporal values, one specifies its valid time and the other is the transaction time, set upon a successful termination of a transaction, where a transaction is defined using the classical definition (e.g. [29]) and refined in the following section. To ensure durability, the values that are changed by the transaction persist after a commit command is issued. It is worth noting that all the state-elements that were generated by a single transaction share the same transaction time.

Information retrieval from a database is done by retrieving state-elements that persist in the database. The following parameters define the set of state-elements which are considered for retrieval:

1. The required variable(s), i.e. a specific object and a list of properties.
2. A temporal element that specifies the required valid time.
3. A chronon that specifies an observation time of the query. An observation time defines a previous state of the database, rather than the current one, to be the retrieved state. A selection of an observation time to be $t_0 < now$ results in selecting only state-elements that are known at t_o, i.e. have persisted in the database no later than t_o ($\{se \mid se.x \leq t_o\}$). It is worth noting that the observation time is restricted to be less than the chronon in which a query was issued. A related type of queries retrieves previously inserted values of a variable $\alpha.P$ in τ. Hence, instead of specifying an observation time, a version number is utilized. For example: "retrieve the value of $\alpha.P$ in τ that was inserted before i versions" ($i > 0$). We term these queries *version queries*.

Let Q be a query for a variable $\alpha.P$ in τ as of t_o. Q is formalized as $Q = \sigma_{X=\alpha.P \wedge v \cap \tau \neq \emptyset \wedge x \leq t_o}(Q')$, where X is the set of attributes that represents the object identifier, and Q' is the query part that is associated with the non-temporal aspects of Q. Q returns the state-elements $\{se_1, ..., se_n\}$ of $\alpha.P$ such that $\forall 1 \leq i \leq n, se_i.v \cap \tau \neq \emptyset \wedge x \leq t_o$. It is worth noting that some queries may require the use of the full set of state-elements of a variable. For example, the query "find the chronons in which the price of a share x increases within a day after an increase in patrol prices," requires the use of the full set of state-elements of the share x and the patrol prices. It is also noteworthy that in order to enable an observation time specification, a participant can be extended to be a triplet $\langle \alpha.P, v, t_o \rangle$ where the latter component represents the observation time.

It is possible to define a preference criterion to provide a partial order among state-elements. A preference criterion that chooses the value(s) that is (are) valid in an interval τ for which a variable $\alpha.P$ has overlapping values can be based on several preference relations. For example, let se_i and se_j be two state-elements of a variable $\alpha.P$ that are candidates for retrieval. se_j is preferred to se_i iff $se_j.x > se_i.x$. This preference criteria (denoted "last value semantics") is the common one in temporal databases and we shall use it as a default criteria. Therefore, a variable for which no observation time is specified (i.e. there is no value specified for t_o) is assumed to require the state-element that has a higher x value for each chronon than of all the other candidate state-elements.

3 A transaction model for temporal databases

In this section we provide a transaction model for temporal databases. Section 3.1 provides the modifications of the basic concepts of transaction modelling. Based on these modifications we provide a temporal transaction model in Section 3.2.

3.1 Modification of basic concepts of transaction modelling

A transaction in a temporal database, just like a transaction in a conventional database, is a set of database operations that the database views as a single unit of work. However, all database operations in a temporal

database are associated with a temporal element that defines their temporal effect on the database (see the definition of a *participant* in Section 2). We limit our discussion to database operations only, although a transaction may consist of external routines,[3] as we are mainly interested in the transaction's effect on the database. In this section we provide the required modifications to transaction modelling in temporal databases.

Atomicity and recovery This section discusses atomicity and recovery. We present temporal independence as a new form of atomicity, and discuss various recovery mechanisms, including transactions aborts, cascading aborts, aggressive and conservative protocols and a redo mechanism.

A transaction in conventional database is *atomic*, i.e. its database operations can either occur in their entirety or not occur at all, and if they occur, nothing else apparently went on during the time of their occurrence [29]. There are many possible temporal extensions to the atomicity property. The two extremes result in two types of atomic behaviour of a transaction, as follows:

Global atomicity: The atomicity as perceived in conventional databases.

Temporal independence: The temporal database is conceptually viewed as a set of independent database snapshots, each of which relates to a different chronon. Hence, a transaction in a temporal database is viewed as a collection of transactions applied to different snapshots, and therefore a transaction can commit in one chronon and abort in another. The effect of temporal independence is materialized in a preprocessing phase, during which a transaction submitted by the user is partitioned into a set of transactions, each relates to a single schema version, that are executed according to a set of algorithms as proposed in [11].

The main discernment for introducing temporal independence is to provide the user with an adequate mechanism to support database operations in a temporal database with schema versioning. A temporal database accommodates *schema versioning* if it supports modifications to the database schema, as well as database operations that should be interpreted in the context of the appropriate schema version, which is not necessarily the current one [18]. The persistence of all schema versions guarantees correct interpretation of historical data, since each update operation o is considered with respect to the schema(ta) that is (are) correct in the valid time as given in the participants of o. Therefore, by using global atomicity, if o cannot be performed with respect to any of the involved schemata, the transaction aborts. Consequently, redundant aborts may occur due to the user's ignorance with respect to the metadata modifications. By using temporal independence, on the other hand, a transaction in a temporal database is treated as a syntactic substitution for representing several snapshot transactions, not bounded by global atomicity rules. Temporal independence, therefore, supports the maximal possible changes to snapshots, while maintaining the database consistency.

To ensure atomicity, the DBMS should use a recovery scheme. Since each state-element is stamped with a transaction time at commit time, the most natural policy to adopt is the *No-steal* policy, according to which no state-elements are written to the database at least until the commitment of the modifying transaction. Therefore, we can assume that all the state-elements that are generated by a transaction persist only at commit time, after a time stamp was chosen. Hence, whenever a transaction T aborts, all the state-elements that were generated by T are not added to the database (no-undo policy). It is possible, that due to various reasons (e.g. shortage of main memory) some of the state-elements are written to the database (using the *Steal* policy), to be replaced at a later time by adding the transaction time. In this case, these state-elements should be erased to ensure a correct recovery process. Since an append-only database simulates the *shadowing* strategy, by keeping all of its previous states, erasing these state-elements restores the previous database state, and ensures the database consistency.[4]

[3] For example, in DB2 a transaction is defined as "a set of interactions between an application and the database." [5]

[4] A less powerful argument, regarding media failures only, was presented in [20].

The problem of cascading rollbacks exists in temporal databases (and can be prevented by using a strict protocol), yet its scope can be narrowed by refining the conflicting operations notion. This refinement also serves to enable a better concurrency of transactions, as discussed in the sequel.

In temporal databases, the occurrence of a deadlock situation is less likely than in conventional databases since temporal databases store and use more information of each property and therefore there is a reduced probability of having two concurrent transactions trying to lock the same item (see section 3.1 for the locking mechanism in temporal databases). Based on results in conventional databases, the use of an aggressive protocol is preferable to the use of a conservative one in temporal databases.

The *redo* mechanism of a conventional database is not adequate for temporal databases. In conventional databases we can scan through the log and update each value of a committed transaction (e.g. [22]). In temporal databases, however, this simple mechanism might generate two similar sets of state-elements in case a system failure occurs while updating the database with the updates of a committed transaction. Such a duplication is likely to affect the database's retrieval results in situations where the number of state-elements is used for view purposes (e.g. averaging the values of a variable at a given chronon). To overcome this problem, we suggest to register the transaction time on the log when a transaction is committed. In the recovery process, the information of a new state-element of a variable $\alpha.P$ will be generated based on the information on the log only if there is no identical state-element in the database for $\alpha.P$. Using this scheme, the state-elements will be recovered as a whole (including the original transaction time) rather than generating a similar set of state-elements.

Temporal locks A common mechanism to ensure serializability of transactions is the locking mechanism. In this section we discuss a refinement of the conventional locking mechanism, to enable a more flexible transaction management, using the unique properties of temporal databases.

Definition 2. - A temporal read lock: A transaction T in a temporal database *holds a temporal read lock from time* t_l *until time* t_u *on a variable* $\alpha.P$ *in* τ (denoted as *trlock* $\langle \alpha.P, \tau \rangle$) iff no transaction can update $\alpha.P$ in τ in the time interval $[t_l, t_u)$.

Definition 3. - A temporal write lock: A transaction T in a temporal database *holds a temporal write lock from time* t_l *until time* t_u *on a variable* $\alpha.P$ *in* τ (denoted as *twlock* $\langle \alpha.P, \tau \rangle$) iff T is the only transaction that can update $\alpha.P$ in τ in the time interval $[t_l, t_u)$.

It is worth noting that there are two different time dimensions in the above definitions. τ is a temporal element that relates to the valid time of a state-element, while $[t_l, t_u)$ defines the time in the real world when other transactions are prohibited from reading/writing $\alpha.P$ in τ. A transaction T can request a *trlock* $\langle \alpha_k.p_l, \tau_q \rangle$ or a *twlock* $\langle \alpha_k.p_l, \tau_q \rangle$. Both types of locks are released with an *unlock* $\langle \alpha_k.p_l, \tau_q \rangle$ request.

As in conventional databases, we assume that each time a *twlock* is applied to a variable $\alpha.P$ in τ, a unique function associated with that lock produces a new state-element for $\alpha.P$ in τ. That function depends on all the variables which were locked using *trlock* prior to the unlocking of $\alpha.P$ in τ. Also, we assume that a *trlock* applied to a variable $\alpha.P$ in τ does not modify $\alpha.P$ in τ. We do not assume, however, that a write lock of a variable implies that it is read.

As a final note, we draw attention to the fact that while in conventional databases a write lock of an element A prevents further read locks to A before the write lock is released, there are situations where a write lock does not prevent a read lock. These situations involve the usage of previous application states using an observation time. Since previous application states cannot be modified in an append-only database, any retrieval operation that involves an application state that precedes the starting of the transaction can be retrieved at any time during the transaction processing, without a need for a lock even if the variable is write-locked at that time [21].

Conflicting operations The common model in conventional databases defines conflicts among read and write operations of the same item, and uses locking as a mechanism to prevent a non-serilaizable schedule as a result of such conflicts. Read locks are considered to be *shared*, i.e. a read lock on an item A prevents any other transaction from writing a new value to A, yet any number of transactions can hold a read lock on A. A write lock, however, is considered to be *exclusive* in the sense that while a transaction holds a write lock on an item A, no other transaction can read from or write to A. As discussed in this section, a refinement of the notion of a conflict is required when discussing temporal databases.

In temporal databases, conflicts may occur among two read or write operations only if they relate to the same variable $\alpha.P$ with an overlapping valid time τ. As discussed in section 3.2, and following similar mechanisms in conventional databases, there can be no RR conflict in temporal databases, yet there exists a WR conflict. However, unlike conventional databases, a WW conflict cannot always be solved by identifying useless transactions[5] in append-only temporal databases. For example, if a transaction uses an observation time to retrieve state-elements from the database, "useless transactions" in a conventional database become "useful transactions" as their values might serve in a future read operation.

Since transactions are time stamped on commit time (x), their effect on further retrievals in the database depends on the order of the transactions' commit commands. For example, if two transactions T_1 and T_2 attempt to write concurrently to a variable $\alpha.P$ values val_1 and val_2 with valid times τ_1 and τ_2, respectively, such that $\tau_1 \cap \tau_2 = \tau \neq \emptyset$, then both values persist, yet only one value is retrieved using the last value semantics. If T_1 commits before T_2, val_2 will be the retrieved value of $\alpha.P$ in τ, and vice versa. It is also possible, under such circumstances, to generate a history that would not be serializable. For example, let T_1 and T_2 be two transactions, and consider the following history:

	T_1	T_2
(1)	write $\langle \alpha.P, \tau_1 \rangle$	
(2)		read $\langle \alpha.P, \tau_1 \rangle$
(3)		write $\langle \alpha.P, \tau_1 \rangle$
(4)		commit
(5)	commit	

Since T_2 is committed before T_1, a serialized execution should be $T_2 \longrightarrow T_1$, and therefore T_2 cannot use the value of $\alpha.P$ in τ as written by T_1. In Section 3.2 we shall show that due to such scenaria, a 2PL protocol cannot guarantee serializability in a temporal database.

3.2 A temporal transaction model

Having defined the required refinements of conventional terminology to support the temporal dimension, this section presents a temporal transaction model using schedules and a temporal serializability test. We use the convention that a serializable schedule of executed operations ensures the consistency and isolation properties, and show that while a 2PL cannot guarantee serializability in bi-temporal databases, a strict 2PL guarantees serializability. We also provide a new protocol, the abort/commit/wait protocol to minimize the number of aborted transactions. In what follows, a transaction is either a transaction as submitted by a user (if using global atomicity) or a transaction as produced by a pre-processing step (if using temporal independence, as defined in Section 3.1).

A schedule $S = \langle a_1, ..., a_n \rangle$ for a set of transactions $T_1, ..., T_m$ is an ordered set of operations of $T_1, ..., T_m$ such that $a_i = T_j$: (tr/tw)lock $\langle \alpha.P, \tau \rangle$ or $a_i = T_j$: unlock $\langle \alpha.P, \tau \rangle$. The following definition defines equivalence of schedules, using the available sets of state-elements.

Definition 4. - Equivalence of schedules : Two schedules S_1 and S_2 are equivalent if:

[5] A useless transaction is a transaction which effect on the database is lost due to later values written to the database [25].

1. For each variable $\alpha.P$, S_1 and S_2 produce similar sets of state-elements.

2. Each temporal read lock of a variable $\alpha.P$ in τ applied by a given transaction occurs in S_1 and S_2 at times when $\alpha.P$ has similar sets of state-elements in τ.

A weaker definition of an equivalence of schedules utilizes the last value semantics as a comparison mechanism, rather than sets similarity. This weaker definition converges to the equivalence definition of schedulers in conventional databases.

As explained in Section 3.1, the granularity of locks in temporal databases involves a temporal element as well as a variable. Therefore, some modifications are required to a precedence graph in order to identify whether a given set of transactions is serializable or not.

Definition 5. - A temporal serilizability graph: Let $S = \langle a_1, ..., a_n \rangle$ be a schedule for a set of transactions $T_1, ..., T_m$. A temporal serilizability graph $G(V, E)$ is a polygraph such that:

- $V = \{T_1, ..., T_m\}$
- E is generated as follows:

  1. **WR conflict:** an edge $\langle \langle T^{'}, T^{''} \rangle, \tau \rangle$ is generated if:

     Write lock: $\exists a_i = T^{'}$: twlock $\langle \alpha.P, \tau^{'} \rangle \wedge$

     Read lcok: $\exists a_j = T^{''}$: trlock $\langle \alpha.P, \tau^{''} \rangle \wedge$

     Write lock precedes Read lock: $i < j \wedge$

     Valid time overlap: $\tau^{'} \cap \tau^{''} = \tau \neq \emptyset \wedge$

     No intermediate conflicting lock: $\forall i < k < j, (a_k \neq T^{*}$:twlock $\langle \alpha.P, \tau^{*} \rangle \vee a_k = T^{*}$:twlock$\langle \alpha.P, \tau^{*} \rangle \wedge$ $\tau^{*} \cap \tau^{''} = \emptyset)$

  2. **WW/RW conflict:** an edge pair $(\langle \langle T^{''}, T^{*} \rangle, \tau \rangle, \langle \langle T^{*}, T^{'} \rangle, \tau \rangle)$ is generated if:

     Existing edge: $\exists \langle \langle T^{'}, T^{''} \rangle, \tau^{'} \rangle \in E \wedge$

     Conflicting item: $\exists \alpha.P \mid (\exists a_i = T^{'}$: twlock $\langle \alpha.P, \tau^{'} \rangle \wedge \exists a_j = T^{''}$: trlock $\langle \alpha.P, \tau^{''} \rangle \wedge i < j \wedge \tau^{'} \cap \tau^{''} =$ $\tau^{'''} \neq \emptyset \wedge$

     Another write lock: $\exists a_k = T^{*}$: twlock $\langle \alpha.P, \tau^{*} \rangle \wedge$ $\tau^{'''} \cap \tau^{*} = \tau \neq \emptyset)$.

According to Definition 5, an edge (or a pair of edges) of the temporal serializability graph connects two transactions only if the destination of the edge can only be performed after the source of the edge. This can occur in the following two situations:

1. A transaction $T^{''}$ reads a value that was written by a transaction $T^{'}$ with an intersecting valid time. Therefore, in a serial schedule $T^{'}$ commits before $T^{''}$.
2. A transaction T^{*} writes a value to a variable $\alpha.P$ in a valid time that intersects with a valid time of $\alpha.P$ that is part of a WR conflict between two transactions $T^{'}$ and $T^{''}$. In this case, T^{*} can commit either before $T^{'}$ or after $T^{''}$.

Definition 5 takes into account the temporal effect, and therefore there should be an overlapping of the locked temporal elements to generate a dependency. It is worth noting that since the retrieval of past application states (using observation times) are not involved in any conflict, they do not require a read lock and therefore do not affect the transactions' priority. However, the order of writing state-elements of the same variable with an overlapping valid time generates a WW conflict. This conflict prevents an erroneous interpretation of version queries.

Definition 6. - A temporal cycle: Let $G(V,E)$ be a temporal serializabiliy graph and let $G^{'}$ be a graph that is derived from G by choosing a single edge of each pair. A *temporal cycle* in $G^{'}$ is a sequence $\langle\langle\langle T^1,T^2\rangle,\tau^1\rangle,\langle\langle T^2,T^3\rangle,\tau^2\rangle,...,\langle\langle T^n,T^1\rangle,\tau^n\rangle$ such that $\bigcap_{i=1}^{n}\tau^i\neq\emptyset$.

Theorem 7. - *Let* $T_1,T_2,...,T_m$ *be* m *transactions with transaction times* $x_1,x_2,...,x_m$*, respectively. A schedule* S *for* $T_1,T_2,...,T_m$ *is serializable iff there is a derivative of the temporal serializability graph* $G^{'}(V,E^{'})$*, built using* S *such that:*

1. *For no two transactions* T_i *and* T_j *such that* $x_i<x_j$, $\langle\langle T_j,T_i\rangle,\tau\rangle\in E^{'}$.
2. $G^{'}(V,E^{'})$ *has no temporal cycles.*

Sketch of proof:[6]

$\Rightarrow$Assume that S is a serializable schedule, yet for any derivative of the temporal serializability graph $G^{'}(V,E^{'})$, built using S, there exist two transactions T_i and T_j such that $x_i<x_j$ and $\langle\langle T_j,T_i\rangle,\tau\rangle\in E^{'}$. Let $\langle\langle T_j,T_i\rangle,\tau\rangle$ be an edge of a derivative of a temporal serializability graph:

1. $\langle\langle T_j,T_i\rangle,\tau\rangle$ was generated due to a WR conflict. $\Longrightarrow$ T_i reads a value that was written by T_j.
   $\Longrightarrow$ T_j should commit before T_i in any serial schedule equivalent to S.
   $\Longrightarrow$ $x_j<x_i$. contradiction to the assumption. **(1)**
2. $\langle\langle T_j,T_i\rangle,\tau\rangle$ was generated due to a WW/RW conflict. $\Longrightarrow$:
   (a) T_j writes a value before T_i and there is some transaction T that reads the value written by T_i.
   $\Longrightarrow$ T_j should commit before T_i in any serial schedule equivalent to S.
   $\Longrightarrow$ $x_j<x_i$. contradiction to the assumption. **(2)**
   (b) T_i writes a value after T_j reads a value written by some transaction T.
   $\Longrightarrow$ T_j should commit before T_i in any serial schedule equivalent to S.
   $\Longrightarrow$ $x_j<x_i$. contradiction to the assumption. **(3)**

(1),(2),(3)$\Longrightarrow$no two transactions T_i and T_j exist, such that $x_i<x_j$, $\langle\langle T_j,T_i\rangle,\tau\rangle\in E^{'}$.

The proof of the second part is similar to the classic proof regarding cycles in a serializability graph (see [29] for an example).

$\Leftarrow$Assume conditions 1 and 2 hold, and assume (without loss of generality) that $x_1<x_2<...<x_m$. Let $R=T_1\rightarrow T_2\rightarrow...\rightarrow T_m$ be a serial scheduler. Using induction, we can show that T_i reads similar sets of state-elements for each variable it locks, both in the given schedule S and in the serial schedule R. The reason being that if transaction T_i reads a value of an item $\langle\alpha.P,\tau\rangle$, then in both schedules the same transactions $T_{j_1},T_{j_2},...,T_{j_k}$ $(1\leq j_1,j_2,...,j_k<i)$ were the last to write $\alpha.P$ in some temporal element τ^* such that $\tau^*\cap\tau\neq\emptyset$, or T_i is the first to read $\langle\alpha.P,\tau\rangle$. Otherwise, a temporal cycle would be generated (contradicting

[6] In this paper we present partial proofs. We present the part of the proof that is unique to temporal databases, and leave out the parts whose proof is similar to the proofs of classic theorems in transaction theory.

condition 2). Using the induction assumption we can show that the last transaction to write a variable $\alpha.P$ in a chronon t is the same in schedules S and R, and therefore similar sets of state-elements are generated for each variable.□

A temporal variation of 2PL, termed *temporal 2PL* requires that in any transaction, all (read and write) temporal locks precede all temporal unlocks. A *strict temporal 2PL* requires all temporal locks to be released after a transaction commits. As mentioned in [21], temporal 2PL cannot guarantee serializability. We use the following example to demonstrate this claim. Let T_1 and T_2 be two transactions and consider the following schedule S:

	T_1	T_2
(1)	twlock $\langle \alpha.P, \tau \rangle$	
(2)	unlock $\langle \alpha.P, \tau \rangle$	
(3)		trlock $\langle \alpha.P, \tau \rangle$
(4)		twlock $\langle \alpha.P, \tau \rangle$
(5)		unlock $\langle \alpha.P, \tau \rangle$

Obviously, S obeys the temporal 2PL. Thus, for a serial schedule $S^{'}$ to be equivalent to S, T_1 should precede T_2. However, if T_1 and T_2 commit on x_1 and x_2, respectively and $x_1 > x_2$, in order for a serial schedule $S^{'}$ to be equivalent to S, T_2 should precede T_1. Therefore, S is not necessarily serializable. It should be noted that the equivalent schedule in a conventional database (where $\langle \alpha.P, \tau \rangle$ is replaced by $\alpha.P$) is serializable, whether T_1 commits before T_2 or vice versa. Hence, the temporal 2PL is not sufficiently strict to enforce a specific order of commit commands. However, as the following theorem shows, a strict 2PL can enforce a specific order of commit commands and therefore can guarantee serializability.

Theorem 8. - *Let* $T_1, T_2, ..., T_m$ *be* m *transactions with transaction times* $x_1, x_2, ..., x_m$*, respectively, and let* S *be a schedule for* $T_1, T_2, ..., T_m$*. If* S *obeys strict temporal 2PL, then* S *is serializable.*

Sketch of proof: Let S be a schedule that obeys strict temporal 2PL and assume that S is not serializable. Using Theorem 7, for any derivative of the temporal serializability graph $G^{'}(V, E^{'})$ built using S, the following two scenaria are possible:

1. $G^{'}(V, E^{'})$ has a temporal cycle. A contradiction is reached in a similar fashion to classic proofs (see [29] for an example).
2. $G^{'}(V, E^{'})$ has no temporal cycles, yet there exist two transactions $T^{'}$ and $T^{''}$ such that $x^{'} < x^{''}$ and $\langle \langle T^{''}, T^{'} \rangle, \tau \rangle \in E^{'}$.
   $\Longrightarrow$due to the protocol strictness, $T^{''}$ should release all of its locks before $T^{'}$ can acquire a lock for some participant $\langle \alpha.P, \tau^{'} \rangle$, where $\tau^{'} \cap \tau \neq \emptyset$. Let t be the time $T^{''}$ released all of its locks.
   $\Longrightarrow$due to the protocol strictness, $x^{''} < t$. **(1)**
   Since $T^{'}$ is not completed by the time $T^{''}$ released all of its locks (it should still acquire at least one more lock), $t < x^{'}$. **(2)**
   (1), (2)$\Longrightarrow x^{''} < x^{'}$. contradiction.

$\Longrightarrow$If S obeys strict temporal 2PL, then S is serializable.□

While strict 2PL ensures serializability, it is not necessarily the best protocol as it reduces concurrent activities. Thus, we present a protocol (commit/abort/wait) in Table 1 to increase concurrency while avoiding redundant aborts.

Algorithm 1 provides the relevant activities of transactions during their life cycle. In addition to retrieving and updating the database, transactions lock and unlock variables and update the temporal serializability graph. A transaction that concluded its activities might be forced to wait before committing, due to other

The commit/abort/wait protocol:

On **start transaction** do:

1 generate a new node T_i in the temporal serializability graph
2 execute operations, using temporal 2PL for locking and unlocking and
update the temporal serializability graph according to its definition

On **end transaction** do:

1 release remaining locks obtained by T_i
2 if exists $\langle T, T_i \rangle \in E$ then:
3 wait
4 else:
5 commit

On **commit** do:

1 remove T_i and all edges $\langle T, T' \rangle$ s.t. $T = T_i$ or $T = T'$
2 end wait
3 commit transaction

On **abort** do:

1 release remaining locks obtained by T_i
2 remove T_i and all edges $\langle T, T' \rangle$ s.t. $T = T_i$ or $T = T'$
3 end wait
4 abort transaction

On **end wait** do:

1 if exists $\langle T, T_i \rangle \in E$ then:
2 wait
3 else:
4 commit

Table 1. Annotated listing of Algorithm 1—commit/abort/wait protocol

transactions that precede it in the temporal serializability graph and did not commit yet. It is worth noting that any transaction would either commit or abort eventually, since the temporal 2PL prevents temporal cycles (although it cannot ensure by itself the order of the committing transactions). Also, a transaction that reaches the **end transaction**[7] phase would eventually commit, as nothing can prevent it from doing so (all activities were successful and there are no temporal cycles).

4 Implementing a temporal transaction model

Having shown the temporal transaction model, in this section we provide a scheme of a temporal transaction model, based on the relational data model. We define the notion of a *shadow relation* and utilize it in an algorithm for a strict conservative temporal 2PL.

Various methods were suggested to map a temporal data structure into a relational model, using normalization rules. A possible implementation can use universal relations as discussed in [24]. Another possible implementation uses the ENF (Extension Normal Form) [12], which is an extension of the TNF (Time Normal Form) [23], as follows. Each relation designates a set of *synchronous attributes*, which are attributes that have common state-element's temporal information (i.e. x and v) at any chronon. Therefore, each relation

[7] The term **end transaction** bares similarity to the term *prepare-to-commit* in distributed database systems (e.g. [15]). We refrain from using this term to avoid confusion.

is augmented with an attribute that represents x and two attributes (v_s and v_e) for the boundaries of a v interval. We can assume that if R is a relation and $X \subseteq R$ is the object identifier, then $X \cup \{x, v_s, v_e\}$ serves as a key for R.

Using ENF, the update of the temporal database is a tuple-based without redundancies. It is worth noting that the representation is restricted since the v can be an interval but not a temporal element. To eliminate this restriction, a separate relation for the v element should be created, identified by a unique state-element identifier and the interval values.

The use of a conventional locking mechanism for a temporal database based on a relational database is impossible. For example, let R be a relation in ENF (where the set $\{a, x, v_s, v_e\}$ serves as a key):

a	b	x	v_s	v_e
a_1	b_1	t_1	t_2	t_4
a_1	b_2	t_2	t_2	t_3
a_1	b_3	t_3	t_3	t_4

Let T_1 be a transaction that requires the locking of the latest value(s) of a variable b of R in [t_2,t_4). Based on a conventional locking mechanism, the first tuple is locked (being the only one with v_s =t_2 and v_e =t_4), while the other two tuples can be accessed. However, using the temporal semantics and assuming that t_1 <t_2 <t_3 <t_4, the last two tuples serve to override the value of the first tuple. Therefore, the last two tuples should be locked, rather than the first one.[8] We suggest to overcome this problem by using the following data structure that maintains the existing locks in the temporal databases at any given time.

Definition 9. - A shadow relation: Let $DBS = \{R_1, R_2, ..., R_m\}$ be a database schema in ENF, such that $\forall 1 \leq i \leq m$, $X_i \cup \{x, v_s, v_e\}$ serves as a key for R_i. A *shadow relation* of a relation R_i is the relation $R_i^{'} = \{X_i, v_s, v_e, lock\}$, where for each property $P_i^j \in \{v_s, v_e\}$, $Dom_{R_i^{'}}(P_i^j) = Dom_{R_i}(P_i^j) \cup \{all\}$.

Definition 9 provides the data structure for the transaction locking mechanism, and uses the notation $Dom_{R_i^{'}}(P_i^j)$ and $Dom_{R_i}(P_i^j)$ to represent the domain of P_i^j in relations $R_i^{'}$ and R_i, respectively. A lock can be associated with either a specific valid time or the full valid time axis (identified by the *all* keyword). Each relation is "shadowed" to provide the information regarding locked items, where a shadow relation of a relation R consists of all the locked intervals of any instance of R. $X_i \cup \{v_s, v_e\}$ consists of sufficient information to identify an object and *lock* identifies a lock as either a "read" or a "write." Each transaction should (read/write) lock a variable in a specified valid time before using it. To identify whether a variable $\alpha.P_j^i$ can be locked in $v^{'} = \{v_s^{'}, v_e^{'}\}$ the following query LQ results in the current lock that is held on $\langle \alpha.P_j^i, v^{'} = \{v_s^{'}, v_e^{'}\} \rangle$: $LQ(\alpha.P_j^i, v^{'} = \{v_s^{'}, v_e^{'}\}) = \pi_{lock}(\sigma_{X_i = \alpha.P_j^i \wedge ((v_s = all \vee v_e = all \vee v_s \leq v_s^{'} \wedge v_e > v_s^{'}) \vee (v_s < v_e^{'} \wedge v_e \geq v_e^{'}) \vee (v_s \geq v_s^{'} \wedge v_e \leq v_e^{'}))}(R_i))$

Table 2 presents an annotated listing of Algorithm 2. The algorithm provides the locking/unlocking mechanism of a strict conservative temporal 2PL. According to the algorithm, a read lock can be assigned with $\alpha.P_j^i$ at $v^{'} = \{v_s^{'}, v_e^{'}\}$ (v can be $\{all, all\}$) only if $LQ(\alpha.P_j^i, v = \{v_s^{'}, v_e^{'}\})$ is either empty or results in a "read" response. A write lock can be assigned with $\alpha.P_j^i$ at $v = \{v_s^{'}, v_e^{'}\}$ only if $LQ(\alpha.P_j^i, v = \{v_s^{'}, v_e^{'}\})$ is empty. The "Wait" statement puts the transaction on a waiting queue for a release of locks. Being a conservative protocol, it prevents deadlocks and can also prevent livelocks under certain conditions.

The size of a shadow relation depends on the number of concurrent running transactions. LQ runs in $O(\lg n)$ in the worst case (where n is the number of locks that currently exist), and therefore adds little overhead to the transaction's performance.

[8] In [30], this problem is handled by using temporary relations, where an interval is replaved by a set of chronons. This solution is far more expensive than the one proposed in this paper.

```
A strict conservative temporal 2PL:
1       For each participant ⟨α.P_j^i, v'⟩ and a lock l do:
2           case LQ(α.P_j^i, v') of
3               ∅:
4                   insert (R_i, α.P_j^i, v', l)
5               {read} :
6                   If l = read then:
7                       insert (R_i, α.P_j^i, v', read)
8                   else:
9                       Wait
10              otherwise:
11                  Wait
12          end /* Case */
13      end /* For */
14      Process
15      For each participant ⟨α.P_j^i, v'⟩ and a lock l do:
16          delete (R_i, α.P_j^i, v', l)
17      end /* For */
```

Table 2. Annotated listing of Algorithm 2—strict conservative temporal 2PL

5 Conclusion

This paper provides a transaction model for temporal databases and presents a new protocol that increases concurrency and reduces redundant abort operations. We also provide a scheme for implementing a temporal transaction protocol on top of a relational database model. The contribution of the paper lies in identifying the unique properties of transaction management in temporal databases and the use of these properties to provide a refined locking mechanism to enhance the concurrency of such databases.

The implementation of the protocols, as provided in this paper, is currently underway at the University of Toronto. Further research would discuss extending the approach suggested in Section 4 by adding temporal support to conventional transaction models, rather than replacing conventional transaction models. Existing protocols should be compared and evaluated to identify the impact of a temporal extension on the performance of transaction management. This combination would enable the use of temporal oriented data along with conventional data within a single database. Such research should provide a robust mechanism to support temporal oriented data in existing databases. Another possible extension of this paper, that the temporal research area might benefit from, consists of using a multilevel transaction model [31] to model temporal transactions.

Acknowledgments

I would like to thank Opher Etzion, Arie Segev and Dov Dori for their collaboration in designing the temporal data model. I would also like to thank the anonymous reviewers and the participants of the Dagstuhl seminar for their remarks and contribution.

References

1. G. Ariav. A temporally oriented data model. *ACM Transactions on Database Systems*, 11(4):499–527, Dec. 1986.

2. A. Bernstein and N. Goodman. Timestamped-based algorithms for concurrency control in distributed database systems. In *Proceedings of the International Conference on VLDB*, pages 285–300, 1980.
3. G. Bhargava and S. K. Gadia. Relational database systems with zero information loss. *IEEE Transactions on Knowledge and Data Engineering*, 5(1):76–87, Feb. 1993.
4. J. Blakeley. Challenges for research on temporal databases. In *Proceedings of the International Workshop on an Infrastructure for Temporal Database*, June 1993.
5. D. Chamberlin. *Using The New DB2, IBM's Object-Relational Database System.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.
6. P. Chrysanthis and K. Ramamritham. ACTA: The saga continues. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 10, pages 349–397. ACM Press. Morgan Kaufmann publisher, Inc., 1992.
7. J. Clifford and A. U. Tansel. On an algebra for historical relational databases: two views. In *Proceedings of the ACM SIGMOD*, pages 247–265, May 1985.
8. P. Dadam, V. Y. Lum, and H.-D. Werner. Integration of time versions into a relational database system. In *Proceedings of the International Conference on VLDB*, pages 509–522, Singapore, 1984.
9. S. Gadia. The role of temporal elements in temporal databases. *Data Engineering Bulletin*, 7:197–203, 1988.
10. A. Gal. *TALE — A Temporal Active Language and Execution Model.* PhD thesis, Technion—Israel Institute of Technology, Technion City, Haifa, Israel, May 1995. Available through the author's WWW home page, http://www.cs.toronto.edu/~avigal.
11. A. Gal and O. Etzion. Parallel execution model for updating temporal databases. *International Journal of Computer Systems Science and Engineering*, 12(5):317–327, Sept. 1997.
12. 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.
13. A. Gal, O. Etzion, and A. Segev. TALE — a temporal active language and execution model. In P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors, *Advanced Information Systems Engineering*, pages 60–81. Springer, May 1996.
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. D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth. Using tickets to enforce the serializability of multidatabase transactions. *IEEE Transactions on Knowledge and Data Engineering*, 6(1), 1994.
16. T. Hadzilacos and C. Papadimitriou. Some algorithmic aspects of multiversion concurrency control. In *Proc. Fourth ACM Sym. on Principles of Database Systems*, pages 96–104, 1985.
17. C. Jensen, J. Clifford, S. Gadia, A. Segev, and R. Snodgrass. A glossary of temporal database concepts. *ACM SIGMOD Record*, 21(3):35–43, 1992.
18. C. Jensen et al. A consensus glossary of temporal database concepts. *ACM SIGMOD Record*, 23(1):52–63, 1994.
19. A. Kumar and M. Stonebraker. Performance evaluation of an operating system transaction manager. In *Proceedings of the International Conference on VLDB*, pages 473–481, Brighton, England, 1987.
20. D. Lomet. Grow and post index trees: Role, techniques and future potential. In *Proceedings of the Second Symposium on Large Spatial Databases*, Zurich, Switzerland, 1991.
21. D. Lomet and B. Salzberg. Transaction-time databases. In *Temporal Databases: Theory, Design, and Implementation*, pages 388–417. Benjamin/Cummings, 1993.
22. M.Stonebraker. Conncurency control and consistency of multiple copies of data in distributed ingres. *IEEE Transaction on Software Engineering*, 3(3):188–194, May 1979.
23. S. Navathe and R. Ahmed. A temporal relational model and a query language. *Information Sciences*, 49:147–175, 1989.
24. B. Nixon et al. Design of a compiler for a semantic data model. Technical Report CSRI-44, Computer Systems Research Institute, University of Toronto, May 1987.
25. C. Papadimitriou, P. Bernstein, and J. R. Jr. Computational problems related to database concurrency control. In *Proceedings of the Conference on Theoretical Computer Science*, 1977.
26. N. Pissinou, R. Snodgrass, R. Elmasri, I. Mumick, M. Ozsu, B. Pernici, A. Segev, and B. Theodoulidis. Towards an infrastructure for temporal databases—A workshop report. *ACM SIGMOD Record*, 23(1):35, 1994.
27. R. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19:35–42, Sep 1986.
28. K. Torp, C. Jensen, and M. Bohlen. Layered temporal DBMS: concepts and techniques. In *Proceedings of the Fifth International Conference On Database Systems For Advanced Applications (DASFAA '97)*, Melbourne, Australia, Apr. 1997.
29. J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Rockville, Maryland, 1 edition, 1988.
30. C. Vassilakis, N. Lorentzos, and P. Georgiadis. Transaction support in temporal DBMS. In J. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases*, pages 255–271. Springer, 1995.

31. G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems (TODS)*, 16(1):132–180, 1991.

This article was processed using the LaTeX macro package with LLNCS style