Concurrency Control for Perceivedly Instantaneous Transactions in Valid-Time Databases

Marcelo Finger*
Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo
mfinger@ime.usp.br

Peter McBrien
Dept. of Computer Science
King's College London
Strand, London WC2R 2LS
pjm@dcs.kcl.ac.uk

Abstract

Although temporal databases have received considerable attention as a topic for research, little work in the area has paid attention to the concurrency control mechanisms that might be employed in temporal databases. This paper describes how the notion of the current time *— also called 'now' — in valid-time databases can cause standard serialisation theory to give what are at least unintuitive results, if not actually incorrect results. The paper then describes two modifications to standard serialisation theory which correct the behaviour to give what we term* perceivably instantaneous transactions*: transactions where serialising T_1 and T_2 as $[T_1, T_2]$ always implies that the current time seen by T_1 is less than or equal to the current time seen by T_2.*

1 Introduction

Query languages for valid-time temporal database normally contain a notion of 'current-time' [1, 2, 3, 4], usually represented as the value of a special variable now. While it is agreed that the value of now should remain constant during a valid-time database operation such as querying and updating [5, 4], no such agreement exists with respect to the behaviour of now during a valid-time transaction.

The problem is that now *cannot be considered as regular data*. Otherwise, a long temporal transaction could read-lock it, preventing the system clock from changing (supposing, for example, that there is a *tick* transaction that models the periodical updates of now). It is highly undesirable that the duration of transactions influences the way other transactions perceive the clock.

*Partially supported by Brazilian CNPq, research grant PQ 300597/95-9.

So now has to be treated as a special value in valid-time temporal databases that demands special treatment. Several distinct possible semantics for now in valid-time transactions have been studied in [6]; the basic choices and their associated problems are presented in Section 2. The basic conclusion drawn in [6] is that there is no 'perfect' semantics for now in valid-time databases. However, we claim that *perceivedly instantaneous transactions* provide the most intuitive semantics for now. In this kind of transaction, now remains constant and its value is determined by the transactions' submission time.

In this paper, we deal with the problem that perceivedly instantaneous transactions bring to valid-time databases, namely that their concurrent execution cannot always be serialised in the standard way. So, after reviewing the possible semantics for now in valid-time databases, Section 2 explains our choice of perceived instantaneity. Section 3 shows that this kind of transaction does not respect normal serialisation theory; it then formulates a *temporal serialisation theory* and characterises temporally serialisable executions (Theorem 3.1). Section 4 proposes two extensions of traditional two-phase locking that guarantee temporal serialisation, namely *maturity ordering* (MO-2PL) and *maturity ordering with resource knowledge* (MORK-2PL); temporal serialisation theory is used to prove their correctness.

2 An Overview of the Semantics of 'Current-Time' in Valid-Time Transactions

There are several possible ways of determining the value of the variable now. Each possible choice for the semantics of now will be named by subscripting the 'pure' variable now with an appropriate letter.

The first semantical choice for temporal transaction's current-time relates to whether the value of now should be user-defined or run-time determined. If it is user defined (now_U), then prior to a transaction submission the user has to provide a value for now_U; in this way, the transaction can be executed as if now_U were any time in history; the value of now_U does not change during execution, unless there is an explicit operation to do that in the transaction body. We do not rule out the existence of transactions with user-defined values for now, but we also want to allow for the possibility to set the value of now automatically.

The intuitive semantics for run-time determined now_R is 'at the time of transaction execution', but here too there are several choices. The crucial one is whether now_R changes or not. If it is allowed to change by mirroring the value of a real-time system clock, then it has been shown that the result of a transaction can be affected by the number of clock ticks occurring during a transaction [6]. This could be interpreted as a violation of the isolation principle of the ACID transaction properties [7].[1] Perhaps more seriously, allowing the value of now to change during a transaction execution puts a heavy burden on transaction programmers, who in that case have to cope with the possibility of clock changes between each pair of now-dependent data accesses. Those problems tell us that time-varying now_R should be considered only in very particular cases and should not be the default semantical choice for now in temporal databases.

So, if the value of now is to be made constant and run-time dependent, what determines its value? Three transaction events can be the determiner:

- *commit time* (now_C): this value is not known during transaction execution. Although updates can be deferred until commit time is known, it is not possible to execute queries that depend on the unknown value of now. It is therefore ruled out.
- *submission time* (now_S): this is the time when the transaction is submitted.
- *begin time* (now_B): due to system load, the actual start of execution is delayed for an arbitrary period after submission.

We show in Section 3 that both begin time and submission time *may fail to serialise* under two-phase locking if we impose that the order of now should be kept by serialisation.

To summarise, as concluded in [6], there are no semantics for the automatic determination of now that are technically simple to achieve and intuitive. While it should always be possible for the user to set the value of now, we propose that the most intuitive semantics for now is a constant one, related in some way to the time of execution of the transaction: *perceived instantaneity* is akin to transaction *atomicity* and frees a programmer from imagining what happens if time changes during execution; furthermore its value should be determined by submission time: it approximates best the semantics of now as 'at this very moment' from the point of view of a user, while begin time is unknown and can only be interpreted as 'as soon as it is possible'. The former is what we call *perceivedly instantaneous transaction* in a valid-time database.

The rest of this paper deals with the concurrency control problems that can arise from our choice, and how to solve them. Section 3 formalises our notion of *perceivedly instantaneous transaction* as an additional restriction on serialisation graphs, and Section 4 presents two concurrency control mechanisms which meet this restriction.

3 Temporal Serialisation

Figure 1 shows two transactions, T_1 and T_2, running concurrently according to the two-phase locking (2PL) concurrency control mechanism.

Transaction T_1 was submitted and immediately started; the reading of the system clock gave t, so $\mathsf{now}_1 = t$. Transaction T_2, on the other hand, perceives $\mathsf{now}_2 = t + 1$. All read and write operations in T_1 and T_2 refer to the perceived current time (now). T_1 starts by reading the value of y, and T_2 starts by reading the values of x; soon after that T_1 wants to write to x but is blocked by the locking system until after T_2 commits. If we want a serialisation of that concurrent execution, then $[T_1, T_2]$ is ruled out by the locking mechanism, so we are left with $[T_2, T_1]$. But if transactions were executed serially in this order, the determination using submission time of the perceived value of now_S would imply $\mathsf{now}_2 \leq \mathsf{now}_1$, contradicting the scenario in Figure 1 where $\mathsf{now}_1 < \mathsf{now}_2$. Note that this 'time going backwards' phenomenon, caused by the choice of submission time determining now_S, can also occur if instead we use now_B (determined by begin time). For example, $\mathsf{now}_B = \mathsf{now}_S$ when the delay between transaction submission and the beginning

[1] In fact, the isolation principle requires the execution of transactions to be independent from each other; here we are assuming a stronger isolation, namely that of external events such as clock ticks; that is why we carefully stated that this *could be seen as* a violation, instead of stating it *certainly is* one.

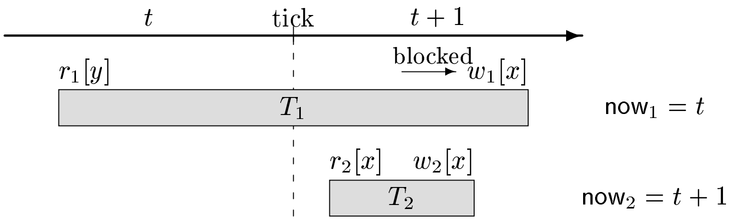


Figure 1: Violation of temporal serialisation

of its execution is null. Time moving backwards even occurs with now_U, but then it would be argued that this would be expected, since the progression of values of now_U would be a matter of user choice.

It follows that normal serialisation does not guarantee a 'now-ordering' which we call *temporal serialisation*. This may cause problems because T_1 is perceived as happening before T_2, so the past data read by T_2 is incorrect according to such perception. Note that adopting a timestamp order concurrency control would not solve the problem: T_1 would be aborted and restarted in its attempt to write to x; its submission time does not change after restart, resulting in the same serialisation $[T_2, T_1]$ but still $\mathsf{now}_1 < \mathsf{now}_2$. If begin time determined the value of now, then after a restart we could have a new value for $\mathsf{now}_1 \leq \mathsf{now}_2$, thus obtaining temporal serialisation. This, however, goes against the intuition of perceived instantaneity. The solution is to try to impose temporal serialisability by means of the concurrency control mechanism. It is this option that we investigate next.

3.1 Temporal Serialisation Theory

We now detail a temporal serialisation theory which may be used to avoid the time moving backwards problem, regardless of the choice of how now is determined. We call the value of now as seen by a transaction the *maturity* of the transaction, and it can be seen as having an integer value. A transaction T_i is more *mature* than a transaction T_j if the value of the current valid time seen by T_i is strictly smaller than that seen by T_j; we represent that by writing $\mathsf{now}_i < \mathsf{now}_j$. Our serialisation theory will ensure that if $\mathsf{now}_i < \mathsf{now}_j$ then T_i will always be serialised before T_j.

A database is seen as a set of 'objects' in the loose sense (not the object oriented sense). The set of possible operations $\mathcal{O}_i$ for a transaction T_i contains, for every object x in the database, $r_i[x]$ (read x) and $w_i[x]$ (write x); it also contains the terminating operations c_i (commit), a_i (abort).

Two operations performed by distinct transactions over the same data x are said to *conflict* if one of them is a write operation. A transaction T_i is then formalised as a partial order $(T_i, \prec_i)$ where:

- $T_i \subset \mathcal{O}_i$;
- either $a_i \in T_i$ or $c_i \in T_i$;
- if l_i is a_i or c_i, then for any other operation $o \in T_i$, $o \prec_i l_i$, i.e. l_i must be the last operation in the transaction;
- if $r_i[x], w_i[x] \in T_i$, then $r_i[x] \prec_i w_i[x]$ or $w_i[x] \prec_i r_i[x]$.

For simplicity and to keep notation unambiguous, it is assumed that at most a single operation on some object x is performed during a transaction, as is usual in classical serialisation theory [8]; however, none of the results in this paper depends on such a restriction.

If $T = \{T_1, \ldots, T_n\}$ is a set of transactions, a *complete history* H^* over T is a partial order $(H^*, \prec_{H^*})$ such that:

- $H^* = \bigcup_{i=1}^{n} T_i$;
- $\prec_{H^*} \supseteq \bigcup_{i=1}^{n} \prec_i$, i.e. the order of H^* is an extension of the orders of T_i's;
- for every two conflicting operations p and q in H^*, either $p \prec_{H^*} q$ or $q \prec_{H^*} p$.

A *history* H over T is simply a prefix of a complete history over T. The *committed projection* of a history H, $C(H)$, is obtained by deleting from H all operations from transactions that are not committed in H.

Two histories $(H, \prec_H)$ and $(H', \prec_{H'})$ are equivalent, which is denoted by $H \equiv H'$, if:

- $H = H' = \bigcup_{i=1}^{n} T_i$, i.e. both H and H' are histories over T;
- For every pair of non-aborted transactions T_i and T_j in T, if p_i conflicts with q_j in H then $p_i \prec_H q_j$ iff $p_i \prec_{H'} q_j$.

A history H is *serial* if for every $T_i, T_j \in H$, all operations of T_i appear before all operations of T_j or vice-versa. A history H is *serialisable* if $C(H)$ is equivalent to some serial history. For non-temporal transactions, the notion of serialisability is all that is needed. But for temporal transactions we do not want more mature transactions to be serialised after less mature ones. Therefore we introduce the notion of temporally preserving histories. A history H is *temporally preserving* iff for every $p_i, q_j \in H$, if $p_i \prec_H q_j$, then $\mathsf{now}_i \leq \mathsf{now}_j$. Finally, we say that a history is *temporally serialisable* (TSR) iff $C(H)$ is equivalent to some history that is both serial and temporally preserving.

Temporal serialisation allows for transactions with the same maturity to be serialised in any order among themselves. But it imposes the restriction that more mature transactions be serialised before less mature ones. For non-temporal transactions it is widely known that serialisability is equivalent to having an acyclic serialisation graph [8]. A similar property applies to histories of temporally serialisable transactions.

Let H be a history over transactions $T = \{T_1, \ldots, T_n\}$, and $TC \subseteq T$ be the set of committed transactions in H. The *serialisation graph* for H, $SG(H)$, is a directed graph whose nodes are $T_i \in TC$ and whose edges are such that $T_i \rightarrow T_j$ iff there exists conflicting p_i, q_j in H such that $p_i \prec_H q_j$. $SG(H)$ is *monotonic* iff $T_i \rightarrow T_j$ implies $\mathsf{now}_i \leq \mathsf{now}_j$. The temporal version of the serialisability theorem is the following.

Theorem 3.1 (Temporal Serialisability) *A history H is temporally serialisable if and only if $SG(H)$ is acyclic and monotonic.*

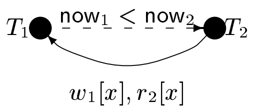
Proof ($\Rightarrow$) Suppose H is temporally serialisable. Then, by classical serialisability theorem, $SG(H)$ is acyclic. Suppose $T_i \rightarrow T_j$ is an edge of $SG(H)$. Then there are conflicting p_i and q_j such that $p_i \prec_H q_j$. Since H is temporally preserving, it follows that $\mathsf{now}_i \leq \mathsf{now}_j$, so $SG(H)$ is monotonic, as required.

($\Leftarrow$) Suppose $SG(H)$ is now acyclic and monotonic, and suppose the nodes of $SG(H)$ are $TC = T_1, \ldots, T_m$. Extend $SG(H)$ into SG_H by adding an edge $T_i \rightarrow T_j$ if $\mathsf{now}_i < \mathsf{now}_j$.

Clearly SG_H is monotonic. We claim it is also acyclic. Since SG_H is monotonic, for every $T_i \rightarrow T_j$ in SG_H we have $\mathsf{now}_i \leq \mathsf{now}_j$. Therefore, for every path $T_{p_1}, \ldots, T_{p_r}$ if T_{p_i} precedes T_{p_j} in the path, then $\mathsf{now}_{p_i} \leq \mathsf{now}_{p_j}$. If there is a cycle $T_{p_1}, \ldots, T_{p_r}, T_{p_1}$ in SG_H, then there must be an extra edge $T_{p_i} \rightarrow T_{p_j}$ in SG_H but not in $SG(H)$ such that $\mathsf{now}_{p_i} < \mathsf{now}_{p_j}$. Since there is also a path from T_{p_j} to T_{p_i}, it follows that $\mathsf{now}_{p_j} \leq \mathsf{now}_{p_i}$, which is a contradiction.

Since SG_H is a directed acyclic graph it may be topologically sorted. Let $T_{s_1}, \ldots, T_{s_m}$ be a topological sort of SG_H, and let H_s be the serial history formed by concatenating the histories of $T_{s_1}, \ldots, T_{s_m}$. Clearly both H_s and $C(H)$ are defined over TC. Suppose there are $T_i, T_j \in TC$ such that $p_i \in T_i$ conflicts with $q_j \in T_j$. If $p_i \prec_{C(H)} q_j$ then there is an edge $T_i \rightarrow T_j$ in SG_H; therefore, in the topological sort of SG_H, T_i must appear before T_j so $p_i \prec_{H_s} q_j$. Conversely, if $p_i \prec_{H_s} q_j$ then either $p_i \prec_{C(H)} q_j$ or $q_j \prec_{C(H)} p_i$ because p_i and q_j conflict; but the latter leads to the contradiction that $q_j \prec_{H_s} p_i$. So $C(H) \equiv H_s$, and hence H is serialisable. Finally, suppose p_i and q_j conflict and $p_i \prec_H q_j$. If $\mathsf{now}_j < \mathsf{now}_i$ then $T_j \rightarrow T_i$ is an edge of SG_H, T_j appears before T_i and, from $C(H) \equiv H_s$, $q_j \prec_{C(H)} p_i$ which contradicts $p_i \prec_H q_j$. So $\mathsf{now}_i \leq \mathsf{now}_j$ of all $p_i \prec_H q_j$ and H_s is temporally preserving. So H is temporally serialisable, finishing the proof. $\square$

Example 3.1 A non-monotonic graph The situation depicted in Figure 1 leads to a serialisation graph that is correct for standard serialisation theory; which would indicate that the only dependency is $T_2 \rightarrow T_1$ due to the conflict between $w_1[x]$ and $r_2[x]$.



The graph is non-monotonic since we can add a dashed line to indicate $\mathsf{now}_1 < \mathsf{now}_2$, and see that the dependency $T_2 \rightarrow T_1$ breaks the monotonicity rule. $\square$

4 Achieving Temporal Serialisation

Having defined a temporal serialisation theory, we now aim to define a concurrency control mechanism which achieves temporal serialisability. We note in passing that conservative 2PL [8] can achieve our aim when using now_B, since the locks and the value of now are determined simultaneously. However, conservative locking is not very efficient, and this is not a general solution for all types of now. Our approach extends the strict 2PL concurrency control mechanism to achieve temporal serialisation. 2PL already guarantees serialisation [8], so we only have to enforce that all temporal histories are temporally preserving.

In 2PL a transaction T_i can perform an operation $p_i[x]$ only after the corresponding lock on x, $pl_i[x]$, has been obtained. Two locks $pl_i[x]$ and $ql_j[x]$ *conflict* if the corresponding operations $p_i[x]$ and $q_j[x]$ conflict.

A lock $pl_i[x]$ is obtained if no conflicting lock on x is being held. In strict 2PL, locks can be released only after the transaction commits or aborts. Then following two protocols develop upon this basic mechanism.

4.1 Maturity Ordering

To achieve temporal serialisation we extend strict 2PL with a mechanism that enforces *maturity ordering*, thus obtaining MO-2PL which satisfies the following rules:

1. The strict 2PL rules.

2. Let T_i and T_j be two transactions such that $\mathsf{now}_i < \mathsf{now}_j$. If T_j is a non-terminated transaction that holds or has held a lock for an object x and T_i requests a conflicting lock on x, then T_j is aborted and restarted, and T_i is given the requested lock.

3. A transaction may only be committed after all more mature transactions have been committed.

Note that rule (3) places a restriction on the value that may be assigned to now in a new transaction; that is the value of now in such transactions can not be more mature than the value of now given to any committed transaction.

Theorem 4.1 *MO-2PL histories are temporally serialisable.*

Proof Let H be an MO-2PL history. Since MO-2PL is simply a restriction of the 2PL rules, it follows that $SG(H)$ is acyclic.

To show that $SG(H)$ is monotonic, suppose by contradiction it has an edge $T_i \rightarrow T_j$ such that $\mathsf{now}_j < \mathsf{now}_i$. Since $T_i \rightarrow T_j$ there must be conflicting operations $p_i[x] \in T_i$ and $q_j[x] \in T_j$ such that $p_i[x] \prec_H q_j[x]$. Either T_i commits before lock $ql_j[x]$ is requested, or afterwards. If T_i committed before $ql_j[x]$, rule (3) would be broken, since T_i is less mature than T_j. If T_i committed after $ql_j[x]$, rule (2) would have caused T_i to be aborted, removing the conflict. Thus $\mathsf{now}_j \not< \mathsf{now}_i$ and $SG(H)$ is monotonic.

It follows by Theorem 3.1 that H is temporally serialisable. □

Example 4.1 Use of MO-2PL for Figure 1 If we use MO-2PL the following changes occur to the normal 2PL behaviour. Firstly, when T_2 reaches its end, it is unable to commit, since the more mature transaction T_1 is still executing, and thus T_2 suspends. Secondly, when T_1 attempts to obtain a lock for $w_1[x]$, T_2 is aborted since it holds a conflicting lock on x and is less mature then T_1. After T_1 commits, T_2 can be restarted. □

Rule (3) above places a heavy burden on the system implementing MO-2PL. If there is a single transaction T that lasts for several chronons (i.e. the basic indivisible units of time [9]), then every transaction submitted in the intermediate chronons while T is executing will have its commitment unnecessarily delayed until, at least, the termination of T. So a single long transaction can cause the delay of several potentially small transactions. The number of transactions in the system can increase significantly, leading to a serious decrease in system throughput known as *thrashing*.

For these reasons, MO-2PL should be used in a system only if transactions are guaranteed to terminate within a relatively short period of time, so that at most one tick event (i.e. a single chronon increment) may occur during the execution of any transaction. The unnecessary delays in the commitment of transactions can then be avoided. If this is not the case, then the following MORK-2PL system should be adopted.

4.2 Enriching Maturity Order with Resource Knowledge

To improve on MO-2PL we have to provide the concurrency control system with *a priori* knowledge of the resources (potential locks) each transaction may need. With the existence of such knowledge we expect to eliminate the heavy burden placed by the commit rule (3) of MO-2PL, and hence provide a greater degree of concurrency between the transactions. The predeclaration of resources is defined as follows.

- Prior to start of execution, a transaction must declare each of the potential locks it may require during execution. Not all predeclared locks need be requested during execution. However, if a non-declared lock is requested, the transaction must be aborted.

By requiring that all transactions predeclare their potential locks, we create *Maturity Ordering with Resource Knowledge* (MORK-2PL), which contains MO-2PL's rules (1) and (2) plus a new commit rule:

3′. T_i can commit while a more mature T_j is still running only if T_j has not predeclared a lock that conflicts with any of those obtained by T_i; otherwise T_i waits for the termination of T_j.

A small adaptation in the proof of Theorem 4.1 shows that:

Theorem 4.2 *MORK-2PL histories are temporally serialisable.*

The previous knowledge of each transaction potential resources, which is the price paid for the improved performance of MORK-2PL over MO-2PL, can be obtained by a special compiler/code analyser.

When there are conflicts between locks obtained by one transaction and potential locks for an executing more mature transaction, MORK-2PL behaves just like MO-2PL; when such conflict does not exist, MORK-2PL allows a transaction to commit much earlier than MO-2PL would have allowed. Note that in the case that each transaction predeclares potential locks on all the database, MORK-2PL degenerates into MO-2PL.

5 Conclusion

In this paper we have presented a solution to the *time moving backwards* problem introduced in [6]. This solution was based around the notion of *perceivedly instantaneous transactions* which required the presentation of a temporal serialisation theory. We then presented and proved the correctness of two protocols for concurrency control of perceivedly instantaneous transactions in valid-time databases. Apart from the obvious contribution to the correct use of now in temporal databases, we believe that our work has a useful contribution in the analysis of the use of the CURRENT_TIMESTAMP variable in SQL92 [10]. This variable would appear to share many of the properties of the now variable in temporal databases.

Work related to our approach can be found in [11], where the scheduling of transactions was submitted to chronological constraints involving the order in which transactions had to be executed; these constraints were totally external to the transactions and do not refer to the current-time perceived by transactions. The serialisation problems encountered in this paper and [11] are different in nature; as a result, [11] could not simply extend an existing scheduling mechanism and had to propose a totally different *chrono-scheduler*.

There are several ways in which our work can be continued. First, although we have theoretically shown that normal concurrency control may fail temporal serialisation, we do not know how often such a violation occurs in practical temporal database applications nor how serious its effects may be. If it happens frequently enough, or has dear consequences even if occurring rarely, this would justify the cost of altering concurrency control mechanisms.

Second, if temporal serialisation is to be implemented, we have to devise ways of doing it efficiently, specially in what concerns the manipulation of transaction resource knowledge. Preferably, we would like to be able to add temporal serialisation to an existing 2PL scheduler without interfering with its internal behaviour, *i.e.* by treating it as a black box. We need also to study potential optimisation for transactions which do not access the now value, and thus need not obey temporal serialisation.

Finally, we have to study how perceivedly instantaneous transactions can coexist with transactions with user defined time, or even with transactions supporting other semantics of now that temporal database applications may require.

Acknowledgments

The authors would like to thank the anonymous reviewers for their carefully reading of this paper, and helpful suggestions for improvements.

References

[1] G. Wiederhold, S. Jajodia, and W. Litwin, "Integrating temporal data in a heterogenous environment", In Tansel et al. [3], chapter 22, pp. 563–579.

[2] N. Sarda, "Algebra and query language for a historical data model", *Computer Journal*, vol. 22, no. 1, pp. 11–18, 1990.

[3] A.U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, Eds., *Temporal Databases: Theory, Design and Implementation*, Benjamin/Cummings, 1993.

[4] R.T. Snodgrass, Ed., *The TSQL2 Temporal Query Language*, Kluwer Academic Publishers, 1995.

[5] J. Clifford, C. Dyreson, T. Isakowitz, C.S. Jensen, and R.T. Snodgrass, "On the semantics of "NOW" in temporal databases", Tech. Rep. R-94-2047, Dept. of Mathematics and Computer Science, Aalborg University, November 1994.

[6] M. Finger and P.J. McBrien, "On the semantics of 'current-time' in temporal databases", in *XI*

Brazilian Symposium on Databases (SBBD'96), http://www.dc.ufscar.br/eventos/sbbd96/, October 1996, pp. 324–337.

[7] T. Härder and A. Reuter, "Principles of transaction-oriented database recovery", *ACM Computing Surveys*, vol. 15, no. 4, pp. 287–317, December 1983.

[8] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[9] C.S. Jensen *et al*, "A consensus glossary of temporal database concepts", *SIGMOD Record*, vol. 23, no. 1, pp. 52–64, 1994.

[10] ISO/IEC, "Database language SQL (SQL-92 or SQL2)", Tech. Rep. 9075:1992, ISO/IEC, 1992.

[11] D. Georgakopoulos, M. Rusinkiewicz, and W. Litwin, "Chronological scheduling of transactions with temporal dependencies", *VLDB journal*, vol. 3, no. 1, pp. 1–28, 1994.