Julian Mehnle
julian@mehnle.net

Dr. Philip Bernstein: Concurrency Control

http://files.mehnle.net/studies/philip-bernstein.pdf

Revision A (2001-12-28)

Hauptseminar Informatik
Database Hall of Fame (WS2001)
Prof. R. Bayer, Ph. D.
Prof. Dr. D. Kossmann

CONTENTS

1 Introduction

1.1 Dr. Philip Bernstein

Dr. Philip A. Bernstein [1] has a long history in the field of databases in theory and practice by authoring and co-authoring at least two books [2, 3] and more than 90 articles [4] on the topic. He has been speaking at both the Very Large Data Bases (VLDB) Conferences [5] and ACM SIGMOD Conferences [6] numerous times since 1975.

In the past, Dr. Bernstein has worked as a professor at Harvard University and Wang Institute of Graduate Studies, and as a database architect at Digital Equipment Corporation (DEC) (which merged with Compaq in 1998).

In 1994, Dr. Bernstein left DEC and joined Microsoft Corporation as a database repository architect. In the same year, he was awarded the ACM SIGMOD Innovations Award for "innovative contributions to the development and use of database systems", which "have been reduced to practice and adopted widely in significant use" [7].

1.2 Overview

Dr. Bernstein has been (and is still) playing a major role in the research and evolution of database concurrency control. This work intends to give an introduction into the theory of transactions and serializability, followed by an overview over various mechanisms for concurrency control that have been developed, some of which have been implemented in many database systems so far.

The underlying database system model of the following sections consists of three major cascaded modules: the *transaction manager*, which receives and preprocesses transactions from the outside world; the *scheduler*, which takes the atomic database operations and orders them according to a specific concurrency control algorithm; and finally the *data manager*, which handles the commitment and abortion of transactions and performs operations on the database (indirectly) through some caching mechanism. Here, mostly just the scheduler will be relevant.

2 TRANSACTIONS

2.1 MOTIVATION

Multi-user database systems by definition have the capability to process queries from more than one user. When working with a single-user database system a user usually does not have a problem with waiting for his current query to finish before being able to issue the next query, while on a multi-user database system a user should be allowed to expect *not* having to wait for another user's query to finish before issuing a query of his own. This leads to queries having to be executed in parallel, that is *concurrently*, usually in an interleaved way on a multitasking operating system.

2.2 TRANSACTION ISOLATION

Now, as will be shown in section 3, problems might occur when naively executing programs from separate queries that *read* and *write* data from and to the same storage locations in a database in an interleaved way.

Of course it would be unreasonable to demand that programs not assume their solitariness on the executing database system, since it would be anything from difficult to impossible for programs being run by queries to anticipate exactly when they will get interrupted by other queries, and, even if that was possible, that would require *every single* program to be designed appropriately. A better way is to design just the database system in a way that relieves programs from having to anticipate potential conflicts.

The solution is a concept called *transaction* (hence the symbol T in the forthcoming examples). A transaction encloses the database operations, most commonly *read* and *write*, of a query and *virtually* reduces their effect on the database to a single point in time, thus ensuring that no two queries can conflict with each other. This property of the transaction concept is called *isolation*, because it isolates transactions from the effects of other transactions that are running concurrently.

Transaction isolation can be achieved through a number of mechanisms, and some of them will be discussed in later sections.

2.3 TRANSACTION ATOMICITY

Sometimes a transaction has to be deliberately aborted by the user (or by the running program itself) because it has been determined that the program cannot continue for logical reasons in the current situation. Also, as we will see in later sections, the database system itself might have to abort a transaction to

grant the isolatability of other currently running transactions (but also of the one being aborted!).

In such a situation it is desirable that not only the execution of the transaction be stopped, but also that all effects it has already caused on the database are completely nullified, so to leave the database in a state as if the transaction had never been started. This property of the transaction concept is called *atomicity*, because a transaction cannot be executed in part, but only as a whole (*all-or-nothing*).

To denote whether a transaction completes properly, that is: commits, or gets aborted, the database operations *commit* and *abort* (respectively) will be used to conclude the transaction's list of operations in the following sections. Sometimes, a *start* database operation is explicitly used to initiate a transaction, but this work implies such an operation to be automatically issued together with the transaction's first real database operation.

2.4 TRANSACTION CONSISTENCY

The transaction concept also has two other major connotations, *consistency* and *durability*, in composition with *isolation* and *atomicity* leading to the acronym *ACID*. Transaction *consistency*, from a given transaction's point of view, guarantees the database's logical integrity aside from temporary inconsistencies caused by the transaction itself during its course of execution. It also ensures referential integrity, which is relevant mostly when using the *constraints* and *triggers* concepts in relational databases.

Transaction *durability* protects the database against system failures, which are beyond the scope of concurrency control and are not discussed here.

3 SERIALIZABILITY AND RECOVERABILITY

3.1 OVERVIEW

As section 2 already suggested, the concurrent execution of queries that involve conflicting operations can lead to problems. (Two operations conflict if their order of execution is relevant to their combined effect on the database or to the result returned by one of them.) This section presents various types of problems that can occur, and then a concept that permits avoiding these problems while still maintaining an interleaved way of execution.

3.2 LOST UPDATES

To illustrate one common type of problem, imagine a program that deposits an amount of money into a bank account:

```
procedure deposit(account_no, amount);
begin
    temp := read(Accounts[account_no]);
    temp := temp + amount;
    write(Accounts[account_no], temp);
    commit;
end;
```

Suppose a query T_1 is running that program to deposit 20 credits into account 5, which contained 1000 credits before, when suddenly the database system interrupts T_1 to execute another query T_2 to run the same program to deposit another 100 credits into the same account. This gives the following history:

T_1 : *deposit(5, 20)*	T_2 : *deposit(5, 100)*
read(Accounts[5]): 1000	
	read(Accounts[5]): 1000
	write(Accounts[5], 1100)
	commit
write(Accounts[5], 1020)	
commit	

Obviously, the 100 credits get lost this way because T_1 overwrites T_2's result. The conflict occurred because T_1 erroneously assumed that it correctly knew the current amount of money in account 5 when adding the 20 credits and

writing the result back to the account. This type of problem is called a *lost update*, because the effects of all but the last of the involved updates are lost.

3.3 DIRTY READS

Another type of problem is called the *dirty read*. Imagine another program to transfer an amount of money from one account to another:

```
procedure transfer(account_no1, account_no2, amount);
begin
    temp := read(Accounts[account_no1]);
    temp := temp - amount;
    write(Accounts[account_no1]);
    temp := read(Accounts[account_no2]);
    temp := temp + amount;
    write(Accounts[account_no2]);
    commit;
end;
```

Now suppose a query T_1 is running *transfer* on the two accounts 3 and 5 to transfer 100 credits. Then in the middle of its execution, T_1 is interrupted by another query T_2 to run *deposit* on account 5, depositing 200 credits. But unfortunately, after T_1 has finished, T_2, not having committed yet, decides that for some weird reason the deposit must be aborted (as is shown later, the database system might also have to make this decision), and thus produces this history:

T_1 : *transfer(3, 5, 100)*	T_2 : *deposit(5, 200)*
read(Accounts[3]): 1000	
write(Accounts[3], 900)	
	read(Accounts[5]): 1000
	write(Accounts[5]): 1200
read(Accounts[5]): 1200	
write(Accounts[5], 1300)	
commit	
	abort

Now account 5 clearly has an incorrect balance, because T_1 trusted the initial balance it read from T_2's result to be valid despite the conflicting query T_2 not having committed yet and therefore not having made any guarantee about the validity of its results.

3.4 Non-Repeatable Reads

A conducted read by a query T_1 becomes a *non-repeatable read* as soon as T_1 is interrupted by another query T_2 that modifies the subject of the preceding read operation. This might become a problem if T_1 subsequently reads the same subject again, expecting to get the same results as before, or even if T_1 has remembered the initial read's result in a temporary variable and bases any further actions on the belief that the stored result is still valid.

3.5 Phantom Problems

One more type of problem is the *phantom problem*, which can occur in dynamic databases, that is, when not only existing data is updated in static storage locations in the database, but data is also inserted into, deleted from, or moved within the database. Consider a query T_1 is summing up the balances of all accounts of the bank, but is then interrupted by another query T_2 which deletes an account somewhere in the range of accounts that T_1 has already read. When T_1 finally commits, the total balance of all accounts is not correct, because it still includes the former balance of the account that has been deleted, which could thus be called a *phantom* account. Essentially, in the most common cases, phantom problems can be avoided if operations on the data that describes the structure of the database (directories, indexes, etc.) are treated as normal database operations like *read* and *write* [2/3.7]. To prevent all cases of phantom problems, more complex concepts such as *predicate locking* would be required [2/3.7, 9/2.3].

3.6 Serializability

Of course, both the lost update and dirty read problems in the above examples (and the other problems as well) would have been avoided had the queries been executed serially, that is one after another. But fortunately, one can find a condition for the correctness of query execution that is much weaker than requiring a plain serial execution.

The idea is, for a given set of transactions, to find an interleaved history that contains every pair of conflicting operations ordered in the same way as, or: is *equivalent* to, a serial history. Obviously, although there are always multiple serial histories, it is sufficient for the interleaved history to be equivalent to *one* of them to ensure correctness. If a history is equivalent to a serial history, it is said to be *serializable*. (This definition holds only for *complete* histories, that is, histories which include an *abort* or *commit* operation for every transaction involved [2/2.2].) More precisely, this concept is called *conflict serializability* because it is based on the notion of ordering conflicting operations in a manner

that produces correct results. While there are other variants of the serializability concept, such as *view serializability* (which is much more expensive to enforce [2/2.6]), they found virtually no use in real-life implementations of database systems. Thus, in the following, *serializable* really means *conflict serializable*.

The histories shown above are not serializable, but for another example, take T_1 is running the *transfer* program to transfer 100 credits from account 3 to account 5, while T_2 is running the same program to transfer 50 credits from account 5 to account 7. A serializable history would be:

T_1 : *transfer(3, 5, 100)*	T_2 : *transfer(5, 7, 50)*
	read(Accounts[5]): 1000
	write(Accounts[5], 950)
read(Accounts[3]): 1000	
write(Accounts[3], 900)	
	read(Accounts[7]): 0
	write(Accounts[7], 50)
	commit
read(Accounts[5]): 950	
write(Accounts[5], 1050)	
commit	

As can easily be verified, this history has the same effect as executing first T_2 and then T_1 in a serial, non-interleaved fashion, and therefore leads to correct results, as all other serializable histories involving T_1 and T_2 would as well.

This shows that it is indeed possible for a database system to do better than just executing queries serially.

3.7 RECOVERABILITY

Next to serializability, as soon as transactions are allowed (or even forced) to abort, there is another aspect of correctness that must be taken into consideration, namely *recoverability*. To understand this, imagine one more program to initialize a set of (newly created) accounts to 0 credits:

```
procedure initialize(account_nos);
begin
    for each account_no in account_nos do
        write(Accounts[account_no], 0);
    commit;
end;
```

Now, suppose a query T_1 runs *initialize* on accounts 8, 9, and 10, but is interrupted by another query T_2 to run *deposit* to store 50 credits on the freshly initialized account 9. This may very well produce a serializable history:

T_1 : *initialize({8, 9, 10})*	T_2 : *deposit(9, 50)*
write(Accounts[8], 0)	
write(Accounts[9], 0)	
	read(Accounts[9]): 0
	write(Accounts[9], 50)
	commit
write(Accounts[10], 0)	
commit	

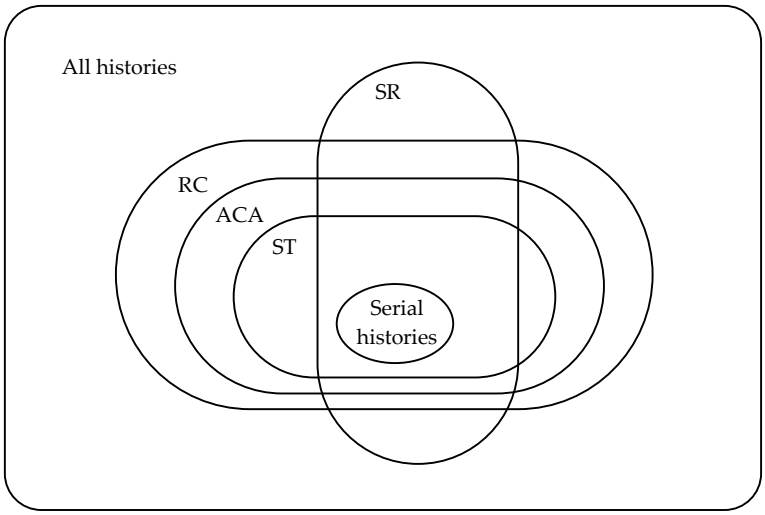
But if, after T_2 committed, T_1 aborted instead of committing as well, it would be unclear whether to restore account 9 to an amount of 50 credits, or 50 credits plus whatever it contained before T_2 started. The situation could not even be recovered by also aborting T_2, because T_2 has already committed and is therefore guaranteed by the database system to persist. So this history is not *recoverable*.

Although this is not a very practical example, it helps to illustrate why it is not enough to enforce serializability alone. In general, recoverability can be achieved by ensuring that no transaction T_i commits until every other transaction T_j that has written data items that are read by T_i has committed itself. This way, if any T_j aborts, the situation can always be remedied by also aborting T_i.

The effect that a transaction's abortion causes the abortion of another transaction is called a *cascading abort*. If a history is very unfortunate, all its transactions could be forced to abort just because a single one transaction was aborted. To avoid the eventuality of cascading aborts, it is required that every data item a transaction *reads* has previously only been written by transactions that are already committed.

Usually, histories are put under an additional but similar restriction, namely that every data item a transaction *writes* has previously only been written by transactions that are already committed. If a history meets this criterion, it is said to be *strict*. Requiring strictness has the benefit that undoing an aborted transaction's effects becomes implementable very easily through a technique called *before images* [2/1.2]. (The before image of a write operation on a given data item is simply the value of the data item before the write operation.)

Taking for granted that aborted transactions can always be restarted, avoiding cascading aborts and strictness are not required for ensuring transaction correctness – both do serve purely practical purposes.



RC: recoverable, ACA: avoiding cascading aborts, ST: strict, SR: serializable

As the above figure shows, serializability and recoverability (including all its sub-properties) are orthogonal, that is neither one implies the other. Anyway, both are required for ensuring transaction correctness in the presence of transaction aborts.

Based on the concepts that have been introduced in this chapter, quite a number of concurrency control algorithms have been developed and implemented in the schedulers of all kinds of database systems. In the following, *two-phase locking* will be explained, which is the most widely-used algorithm. Finally, two alternative mechanisms, *timestamp ordering* and *serialization graph testing*, will be briefly presented as well.

4 TWO-PHASE LOCKING

4.1 BASIC TWO-PHASE LOCKING

The fundamental idea of *two-phase locking*, and for locking in general, is to indicate for every data item whether it is currently being accessed by a transaction (and in what way), so any conflicting transactions can either be allowed to access the same data item concurrently, or be delayed until the accessing transaction has finished. This in-use indication is usually realized through some kind of flag on the data item, called a *lock*, and being kept in a lock table. Basically, every transaction that wants to access a certain data item has to obtain a lock on that item from the database scheduler, and by the time a transaction has terminated, that is, has committed or aborted, it must have released all its locks.

Depending on the available database operations and how they conflict with each other, various types of locks are required. For a given data item, one *read* does not affect another *read*, so multiple reading transactions can share their right to access the data item, but clearly a *write* does affect a *read* or a *write*, thus writing transactions require an exclusive right to access the data item. So, as far as *read* and *write* are concerned, two types of locks are sufficient: *shared locks* and *exclusive locks*. (If a database system supports other operations, such as *increment* or *decrement*, which, for instance, do conflict with *read* and *write* but are compatible with each other, it might be useful to also employ other types of locks [2/3.8].) As the below compatibility matrix depicts, *shared* and *exclusive* lock compatibility is analogous to the compatibility of the *read* and *write* operations:

	S	X
S	✓	-
X	-	-

S: shared lock, X: exclusive lock

Now, a scheduler that employs the basic two-phase locking protocol obeys the following ruleset:

1. Whenever the scheduler receives an operation on a data item from a transaction, the scheduler checks whether another transaction already owns a conflicting lock on that item. If so, the transaction is suspended until all conflicting locks have been released and the required lock can be granted. Otherwise, the scheduler grants the required lock right away. Also, a so-called lock conversion or lock upgrade can happen if the transaction already holds a lock on the data item, but the pending operation re-

quires a stronger lock type. In this case, a shared lock may be upgraded to an exclusive lock, and this counts as an ordinary lock acquisition.

2. No lock will be released before the database data manager has acknowledged that it has processed the operation protected by that lock.
3. Once the scheduler has released *any* lock for a given transaction, it may no longer grant *any* new locks to that transaction.

Rule 1 and 2 ensure that no two transactions can concurrently execute conflicting operations on a data item, and that all conflicting operations are actually processed in the same order in which the corresponding locks are obtained. Rule 3 divides every transaction in two phases (hence the term *two-phase locking*), namely a *growing phase*, in which all required locks are obtained, and a *shrinking phase*, in which all obtained locks are released. This is most important, as it ensures serializability.

To understand this, consider the flawed history to the left, in which *(un)lockS* and *(un)lockX* symbolize the locking and unlocking operations for shared and exclusive locks:

T_1	T_2
lockS(A)	
read(A)	
unlockS(A)	
	lockX(A)
	write(A)
	lockX(B)
	write(B)
	unlockX(A)
	unlockX(B)
	commit
lockX(B)	
write(B)	
unlockX(B)	
commit	

T_3	T_4
lockS(A)	
read(A)	
lockX(B)	
unlockS(A)	
	lockX(A)
	write(A)
write(B)	
unlockX(B)	
commit	
	lockX(B)
	write(B)
	unlockX(A)
	unlockX(B)
	commit

By unlocking data item A too early, T_1 gave T_2 the chance to write A *after* T_1 read A, and then to write B *before* T_1 wrote B. Thus, the history is no longer serializable, since it is unclear whether T_2 has in effect executed after or before T_1 (and of course, neither is really the case).

But with only minor modifications it is possible (and quite easy) to transform the left history into the one to the right, which conforms to the two-phase locking protocol. The right history also shows that rule 3 takes care that all pairs of conflicting operations of the two transactions are scheduled in the same or-

der, that is, T_1 is the first to operate on A, as well as to operate on B. Therefore the history is indeed serializable

4.2 Strict Two-Phase Locking

There is a variant of basic two-phase locking called *strict two-phase locking*, which, for a reason, is in fact how the schedulers of most two-phase locking database systems are implemented. The only difference to the basic two-phase locking protocol is that a transaction's locks are being held until the transaction has committed (or aborted), at which point all of its locks are then released at once. Through this modification of the protocol, strict two-phase locking only produces strict histories, and therefore has the big advantage of being recoverable and avoiding cascading aborts [2/3.5].

4.3 Deadlocks

Unfortunately, two-phase locking in general is prone to the *deadlock* phenomenon which requires the scheduler to abort a transaction for the purpose of maintaining serializability without violating the two-phase locking protocol. This can happen when two transactions are directly or indirectly waiting for each other to release a lock, or when two transactions concurrently try to upgrade their read locks on a data item into write locks [2/3.2].

So in the first place, deadlocks need to be detected. There are a number of ways to do this, and one of the simplest is to attach a *timeout* to each transaction. If a transaction does not terminate before its timeout expires, it may be suspected to be caught in a deadlock, and the scheduler can choose to abort it. The timeout period is a tuning parameter that should be calibrated carefully. Of course, mistakenly aborting a timed-out transaction, although it will hardly ruin the day as the transaction can spontaneously be restarted, is not desirable.

Another way to detect deadlocks is to maintain a *waits-for-graph*. This is a directed graph whose nodes are the transactions that are currently active, that is, neither committed nor aborted. For every transaction waiting for another transaction to release a lock, there is an edge in the graph that points from the waiting transaction to the transaction being waited for. If the graph has one or more cycles, there is a deadlock. Obviously, if the scheduler promptly checks the graph every time it grants a lock, it is always sufficient to remove from the graph, that is: abort, exactly one transaction to break all cycles. Most of the time, the scheduler can even pick one from a row of transactions to abort. Thus, waits-for-graph testing is a good method because it gives the scheduler a lot of flexibility in choosing what transaction is the most effective to abort while not making too much of a waste. On the other hand, the continuous cycle checking is not really a light-weight task [2/3.4].

5 TIMESTAMP ORDERING

Another way for a database scheduler to produce serializable histories is the *timestamp ordering* algorithm. Every transaction that the scheduler receives must have been tagged with a unique timestamp by the transaction manager. Timestamp ordering, like the name suggests, then simply orders all conflicting operations according to their timestamps (equating to the timestamps of their respective transactions), that is, for every pair of conflicting operations, the scheduler forces the data manager to first process the operation with the lower timestamp, and then the operation with the higher timestamp.

To achieve this behavior, the scheduler remembers for every data item the maximum timestamps of all *read* operations and all *write* operations on that item that have been scheduled. Now, according to the type of the operation the scheduler receives, there are two cases:

When the scheduler receives a *read* operation on a given data item, it checks whether its timestamp is lower than the maximum *write* timestamp for that item. If this is the case, the operation cannot be scheduled at all without violating serializability, so the corresponding transaction must be aborted. Otherwise, the operation can be scheduled, and if its timestamp is also higher than the maximum read timestamp, the latter is updated to the current operation's timestamp.

When the scheduler receives a *write* operation on a given data item, it checks whether its timestamp is lower than either the maximum *read* timestamp or the maximum *write* timestamp for that item. If either is the case, the operation again cannot be scheduled at all without violating serializability, so the corresponding transaction must be aborted. Otherwise, the operation can be scheduled, and the maximum write timestamp is updated to the current operation's timestamp (since it is already guaranteed to be obsolete by the above test).

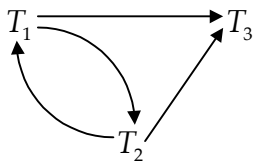
Moreover, it is important to notice that when an operation is scheduled, it still must not be issued to the data manager until the data manager has completed processing all conflicting operations it has received so far. This ensures that all conflicting operations are processed by the data manager in the same order as the scheduler issues them, and thus guarantees serializability.

To additionally guarantee recoverability, and even strictness, a scheduled operation needs to be delayed even longer before being issued to the data manager, namely until all transactions that include any conflicting operations have committed or aborted. This variant is called *strict timestamp ordering*.

6 SERIALIZATION GRAPH TESTING

The last concurrency control method to be covered is *serialization graph testing*. This algorithm maintains a data structure called a *stored serialization graph*, which is a special form of a *serialization graph*. A serialization graph is a directed graph whose nodes are all the committed transactions of a given history. Every edge in the graph points from a transaction T_1 to another transaction T_2 exactly if one of T_1's operations precedes and conflicts with one of T_2's operations in the history. Obviously, if, and only if, a serialization graph is acyclic, it represents a serializable history (for a formal proof, see [2/2.3]). For example, suppose the following history involving transactions T_1, T_2, and T_3:

T_1	T_2	T_3
write(A)		
	write(A)	
	write(B)	
	commit	
write(B)		
		write(A)
		write(B)
		commit
commit		



To the right is the history's serialization graph, which has a cycle involving T_1 and T_2, and thus shows that the history is not serializable.

A stored serialization graph differs from a normal serialization graph only in that it also contains uncommitted (but not aborted) transactions, and transactions that have committed a long time ago may be omitted from the graph under certain premises.

Now, when a serialization graph testing database scheduler receives an operation, it tests whether adding edges (with an appropriate node being created before, if necessary) for all conflicts between the received operation and any previously scheduled operations produces one or more cycles.

If so, the operation cannot be scheduled without violating serializability, so its corresponding transaction is aborted and then removed from the graph (together with all edges connected to it).

Otherwise, the operation can be scheduled. Even so, as in timestamp ordering, the operation still must not be issued to the data manager until the data manager has completed processing all conflicting operations it has received so far. Only then is the history guaranteed to remain serializable.

Again, like in timestamp ordering, strictness can be achieved by requiring all transactions to have committed or aborted that include any operations conflicting with the operation to be issued to the data manager.

Clearly, checking a directed graph for cycles becomes more and more time-consuming when nodes and edges are added to it continuously, so it is desirable to remove unneeded transactions from the graph as soon as possible. For one thing, a transaction cannot be removed from the graph until it has terminated, but it might be necessary to keep the transaction's node and related edges even longer. Only if the transaction is committed and its node has no incoming edges, the node and all outgoing edges that remain can be dropped from the graph. This is due to the fact that while, after a transaction has committed, no new incoming edges can reach its node, it is very possible for a node to grow new outgoing edges. To understand this, from the above example, try to imagine how the stored serialization graph looked when T_1 issued its *write(B)* operation. This was exactly the time when the edge pointing from T_2 to T_1 was introduced into the graph although (or right because) T_2 had already committed.

7 CONCLUSION AND OUTLOOK

The *ACID transaction* concept is very fundamental to most database systems, because it guarantees the user that his work on the database is never affected either by the work of other users on the same database at the same time, or by system failures, which could otherwise jeopardize the integrity and consistency of his data. The theoretical concepts that stand behind transaction isolation, atomicity, and consistency, are *serializability* and *recoverability*. They represent the conditions under which a correct operation is guaranteed.

Luckily, there is a variety of techniques that help to achieve serializability and recoverability in practical implementations. Thanks to simplicity and a relatively low administrative overhead, *two-phase locking* (and especially its strict variant) has stood the test of time and has become the concurrency control algorithm of choice for most transaction-based database systems of today. Anyhow, *timestamp ordering*, which is also quite simple, has found some use, too.

Serialization graph testing, though more complex in implementation, really shines through its beauty, and, above all, does have some potential in allowing much more interleaving and therefore concurrency than other algorithms while still maintaining correctness [2/4.3].

Today, Dr. Philip Bernstein is a senior member of Microsoft's Database Research Group [8] where he is, among other things, still working on improving the comprehension of concurrency control in particular and transaction processing in general [9]. In addition to that, he is doing pioneer work on object databases and database model management. Moreover, Dr. Bernstein is an affiliate professor at University of Washington and holds chairs and membership in an amazing number of database related associations and committees, including the ACM SIGMOD Awards Committee.

BIBLIOGRAPHY

1 Philip Bernstein's Homepage
http://www.research.microsoft.com/~philbe/

2 Philip Bernstein, Vassos Hadzilacos, Nathan Goodman
Concurrency Control and Recovery in Database Systems
Addison-Wesley, 1987
ISBN 0-201-10715-5
http://research.microsoft.com/pubs/ccontrol/

3 Philip Bernstein, Eric Newcomer
Principles of Transaction Processing (for the Systems Professional)
Morgan Kaufmann Publishers, January 1997
ISBN 1-55860-415-4

4 DBLP list of Philip Bernstein's publications
http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/b/Bernstein:Philip_A=.html

5 Very Large Data Bases (VLDB) Conference
http://www.vldb.org/dblp/db/conf/vldb/

6 Association for Computing Machinery (ACM) Special Interest Group on Management of Data (SIGMOD) Conference
http://www.acm.org/sigmod/

7 Recipients of ACM SIGMOD Innovations Awards
http://www.acm.org/sigmod/sigmodinfo/awards.html

8 Microsoft Corporation Database Research Group
http://www.research.microsoft.com/research/db/

9 Hal Berenson, Philip Bernstein, Jim Gray, et al.
A Critique of ANSI SQL Isolation Levels, published in:
Proceedings of the 1995 ACM SIGMOD Conference
ACM Press, 1995
ISBN 0-89791-731-6
http://portal.acm.org/citation.cfm?id=223784.223785
MSR Technical Report 95-51: A Critique of ANSI SQL Isolation Levels
Microsoft Research, June 1995
http://research.microsoft.com/scripts/pubs/view.asp?TR_ID=MSR-TR-95-51