Allocating Isolation Levels to Transactions

Alan Fekete
School of Information Technologies
University of Sydney
fekete@it.usyd.edu.au

ABSTRACT

Serializability is a key property for executions of OLTP systems; without this, integrity constraints on the data can be violated due to concurrent activity. Serializability can be guaranteed regardless of application logic, by using a serializable concurrency control mechanism such as strict two-phase locking (S2PL); however the reduction in concurrency from this is often too great, and so a DBMS offers the DBA the opportunity to use different concurrency control mechanisms for some transactions, if it is safe to do so. However, little theory has existed to decide when it is safe! In this paper, we discuss the problem of taking a collection of transactions, and allocating each to run at an appropriate isolation level (and thus use a particular concurrency control mechanism), while still ensuring that every execution will be conflict serializable. When each transaction can use either S2PL, or snapshot isolation, we characterize exactly the acceptable allocations, and provide a simple graph-based algorithm which determines the weakest acceptable allocation.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Transaction processing*

Keywords

Concurrency Control, Serializability, Anomaly, Consistency, Snapshot Isolation, Two-Phase Locking

1. INTRODUCTION

One major domain where DBMS are used in industry is in on-line transaction processing (OLTP), where fast-acting independently-coded application programs are executed to carry out business activity (for example, modifying inventory and financial and customer data to reflect that a sale has occurred). In such systems, data integrity is very important: this involves both the consistency of data values to the real-world state, and also the validity of constraints (such as referential integrity, or the equality of summary data with the aggregate of the data being summarized) that express internal consistency between parts of the data. There are many threats to the integrity of data in an OLTP system, but one important threat comes from the concurrent activity of separate applications; without careful control of interleaving, called *concurrency control*, anomalies such as Lost Update or Inconsistent Read can lead to data being corrupted.

The abstraction of "ACID transactions", and particularly the understanding of how DBMS mechanisms can prevent concurrency risks to data integrity, were among the great contributions for which Jim Gray won the 1998 Turing Award. These ideas have been expanded into a powerful theory, with two main aspects. One aspect is the identification of those interleavings that are considered correct (because they do not threaten data integrity); the other aspect is in developing (and proving correct) algorithms which control the interleavings.

Serializabilty has long been accepted as the appropriate notion of correctness for executions of a collection of transactions against a database. A serializable execution is one in which each committed transaction sees the same values, and the final state contains the same values, as in a serial or batch execution, where the transactions are run one-by-one with no concurrency at all. The great virtue of a serializable execution is that we can be sure that any integrity constraint holds for the final state of the data, provided that it holds for the initial state, and that each transaction program, when acting alone, from a state where the constraint holds, produces another such state. That is, if the application programs are checked one-by-one, the DBA doesn't need to worry about subtle problems arising from their interleaving.

There are many concurrency control mechanisms in the literature, which can ensure that all executions are serializable (no matter what transactions are run). In practice, however, it is clear that the strict two-phase locking mechanism (with index locking of some variety) is the dominant way to guarantee serializability [6]. Many commercial systems, including IBM's DB2, Microsoft's SQL Server and Sybase's ASE have been built based on this algorithm. Unfortunately, even with all the wrinkles system designers have been able to invent, the performance impact (from the reduced concurrency when executing with two-phase locking) is still seen as heavy, and indeed most transactions on these engines use lower isolation levels, where shared locks are released early or not taken at all.

An alternative multiversion concurrency control approach is found in systems including Oracle and PostgreSQL. This approach avoids taking shared locks at all, by accessing old versions of data (which may be kept anyway for rollback purposes). This algorithm, called Snapshot Isolation (abbreviated as SI), avoids all the well-known concurrency anomalies such as Inconsistent Read and Lost Update. However, as proved in [2], this algorithm does not ensure serializable execution for all transactions[1] and it can lead to violation of integrity constraints. In simplified terms, the Snapshot Isolation concurrency control mechanism means that a transaction T sees the database state produced by all the transactions that committed before T started. Thus if T_1 and T_2 are concurrent transactions using the Snapshot Isolation mechanism, neither will see the effects of the other (whereas in a serial execution, and therefore also in a serializable execution, one of the transactions would see the effects of the other).

In this paper, we consider the problem of how to use the DBMS's capability to support different isolation levels (and thus different concurrency control mechanisms) for different transactions. While practioners frequently do this, there has previously been little theory to guide them in making these decisions wisely.

In particular, we consider a DBA who can choose to use the strict two-phase locking mechanism for some transactions, while other transactions are using the snapshot isolation mechanism. This will be very easy in the forthcoming Yukon release of Microsoft's SQL Server, where the DBA can include the "SET ISOLATION LEVEL" statement in each program before the first other access to the database, and the engine supports both "ISOLATION LEVEL SNAPSHOT" and "ISOLATION LEVEL SERIALIZABLE" (which will use two-phase locking). In Oracle, a transaction preceded by "SET ISOLATION SERIALIZABLE" will use snapshot isolation, but two-phase locking can be obtained by using the default "read committed" isolation and explicitly setting shared locks by "LOCK TABLE name IN SHARE MODE" before reading from a table.

We provide a theory that enables the DBA to decide, for each possible allocation of concurrency mechanism to transactions, whether or not the allocation is acceptable in the sense that all executions that arise will be serializable (and therefore will maintain the integrity of the data). Of course there is always at least one acceptable allocation: use two-phase locking for every transaction. However, as we will show, our theory can prove whether other allocations are also acceptable, and often the DBA will be able to prove that many of the transactions can be allowed to use the snapshot isolation mechanism.

In Section 2 we explain the details of the concurrency control mechanisms involved, and we summarize existing definitions and theory on which we build. In Section 3 we analyze the conflicts in a single execution, and show some properties that hold in any non-serializable execution. Section 4 then looks at a collection of transactions, and an allocation of concurrency control mechanisms to those transactions; our main result is Theorem 3 which characterizes exactly when an allocation has the property that every execution that can arise is conflict-serializable. Section 5 looks at the space of all possible acceptable allocations, and shows operations on allocations that preserve acceptability. Section 6 concludes.

[1]Note that there are multiversion algorithms which ensure serializability in all executions, as shown in ch 5 of [4]. The Snapshot Isolation algorithm is different from these.

1.1 Related Work

There is a huge literature on serializability theory and concurrency control. See [4, 8, 6, 12] for expository accounts and bibliographies. However, most of this prior work considers only situations where correctness is guaranteed without any knowledge of the application or of integrity constraints, which does not apply to SI. The Snapshot Isolation mechanism was first described in the research literature in [2], which also demonstrated that this mechanism does not ensure correctness in all circumstances. Previous work [5] showed ways to prove serializability for certain application mixes when all the programs use SI for concurrency control. Unlike the current paper, this work does not consider mixing transactions running different algorithms.

There is a long research tradition that deals with the mixing of different concurrency control techniques applied to different items among the data, either per object, or per site in a federated system. [11] provided a detailed analysis of when per-object concurrency control algorithms can be combined, but each is required to guarantee local serializability without specific application knowledge (unlike SI). This line of work continued in many papers dealing with federated or multi-database systems; there, as an added complication, the mechanisms may be unknown or incompatible, so one needs extra coordination such as accessing a ticket item. In [10] this approach was applied to federations including some sites using SI, while others used algorithms (such as two-phase locking) that ensured local serializability. Note that all of this work is incomparable to our discussion, where a varying concurrency method is associated with each separate transaction, rather than with separate parts of the data.

A quite different approach to ensuring correctness when mixing isolation levels is in [3], which uses specific knowledge of the integrity constraints that are required, and the condition proved is that the given constraint is preserved, rather than more general serializability as in this paper.

2. BACKGROUND MATERIAL

In this section we summarize the standard concepts and definitions, on which we will build our results.

2.1 Combinatorics definitions

We collect here definitions we use of concepts from discrete mathematics, especially combinatorics. The concepts are all wide-spread, but terminology varies widely, so we give definitions to prevent confusion.

A directed graph consists of a set of *nodes* and a set of *edges*. Each edge is defined as an ordered pair of nodes, called the *source* and *target* of the edge. Thus any binary relation on a set can be used to define a directed graph, whose edges are the pairs in the relation.

In a *labelled directed graph*, each edge is associated to a non-empty subset of the labels. Thus the set of edges associated to a given label defines a subgraph of the original directed graph. Note that we do allow an edge to have multiple labels.

A directed walk in a directed graph G is an alternating sequence of nodes and edges, v_α, e_α, v_β, e_β, $\ldots$, v_η, such that each edge e has as source the node v which immediately precedes e in the walk, and as target the node v' which immediately follows e. A cycle is a directed walk whose initial node and final node are equal, and where no other repetitions occur among the nodes of the sequence. A directed graph is called *acyclic* if it does not contain a cycle.

A cycle v_α, e_α, v_β, e_β, $\ldots$, v_α in the directed graph G is called *chord-free* if for every pair of nodes (v_ψ and v_ϕ, say) which occur in the cycle and are not joined by an edge in the cycle, then v_ψ and v_ϕ are also not joined in G. Note that our definition allows the case where successive nodes in the chord-free cycle are joined by a reverse edge in G. If a cycle is of minimal length in G, it is necessarily chord-free.

A partially ordered set $(X, \preceq)$ is a set X with a binary relation $\preceq$ on X that is reflexive, antisymmetric and transitive. A partially ordered set is called a *lattice* if every pair of elements x and y have a least upper bound (also called join) $x \vee y$ and a greatest lower bound (also called meet) $x \wedge y$. In a lattice $(X, \preceq)$, a set of the form $\{x \in X : x \preceq a\}$ is called a principal ideal generated by a.

2.2 Concurrency Control Algorithms

The traditional state-of-the-practice for concurrency control has been based on locks [6]. Most products allow each transaction to explicitly set and release some locks, and they also provide implicit mechanisms based on the transaction declaring an isolation level; the DBMS engine then sets locks on various items accessed, and it ensures that conflicting locks can't be held on any item (a request for an unobtainable lock will block until the lock becomes available). There are many subtle issues to ensure acceptable performance, including detecting and handling deadlocks, using locks at several granularities (with appropriate intention modes to keep them coordinated) and also setting locks appropriately when using indices in a predicate read, which calculates which records match the where clause in a query (see section 7.8 of [6]). For full isolation, the key feature is that locks must be held until the transaction completes (this implies that locking obeys the "Two phase" rule where no new locks are obtained by a transaction once it has released any lock). In this paper we will refer to this as Strict Two-Phase Locking (abbreviated as S2PL) concurrency control[2].

The seminal paper [2] attacked the traditional view of isolation definitions, in part by pointing to an algorithm that avoids all the traditional anomalous executions, and yet it allows interleavings that are not serializable, and lead to violation of integrity constraints. This concurrency control algorithm is called Snapshot Isolation (abbreviated as SI), and it has been implemented in much-used DBMS engines, for example in Oracle [7] and PostgreSQL (however, both these products use the algorithm inappropriately for the situation when the application programmer declares that a transaction should be "Serializable"). The forthcoming Yukon release of Microsoft SQL Server will also provide this algorithm, as a weaker alternative isolation level alongside S2PL.

[2]The SQL standard uses the term "SERIALIZABLE" isolation level, and it does not assume that the particular strict two-phase locking algorithm is used. However, our theory depends crucially on the particular mechanism, where commit-duration locks are held, and so we will throughout the paper refer to the S2PL isolation level. We reserve the term Serializable for the standard theoretical definition of an interleaved execution of many transactions, some of which may be declared with different isolation levels, but which are interleaved in a way that see the same values and leave the same final state, as happens in a serial pattern of execution [8, 4, 12]. We note that when every transaction uses the S2PL isolation mechanism, then the whole execution is serializable.

SI is a multiversion mechanism: it keeps several versions of an item around at the same time, and allows a transaction to access other versions as well as the most recent value for the item. In many implementations, keeping the versions is being done anyway, for support of possible rollback events. When a transaction T using the SI mechanism wishes to access an item, it sees the version which was most recently committed at the time T started. (One exception to this is when T has itself issued a modification to the item; in that situation, T sees its own most recent version.) This use of the state as it was when T started, called T's snapshot, applies also to indices when evaluating where clauses. In effect, T's updates are kept in a private universe, not visible to other transactions, until T's commit, when the versions it created become installed and visible to all. In order to avoid lost updates, a transaction T running with the SI mechanism, will not be allowed to commit if it will install a version of some item x which overwrites a version which was installed while T was active (and thus which was not part of T's snapshot). This is called the "First Committer Wins" rule.

While it is not mentioned in [2], implementations of SI such as Oracle's ensure that a version of an item x produced by an SI transaction T must be protected by an exclusive lock from the time it leaves any private universe of T, until (and including the instant when) the version is installed because T commits. The exclusive lock must be obtained following the normal rules for such locks (e.g. intention locks must be obtained on enclosing granules, the lock can't be obtained if a conflicting lock is held by another transaction, etc). We require this, as it ensures that SI and S2PL transactions avoid lost update interleavings with one another. Similarly, the check against overwriting a version which was installed while an SI transaction T was active, will cover versions produced by locking transactions as well as versions produced by SI transactions.

We have described the SI mechanism quite broadly. Our results will also apply to various special cases: for example, some implementations of SI do not use a private universe for an SI transaction T. Instead they have T do its update in-place, with T taking an exclusive lock, and performing the check against overwriting another version, at the time the SI transaction creates the version; then T holds the lock till commit. This has the properties stated above (the lock is held from the time the version enters the public space until T commits, and T doesn't commit if it produces a version that overwrites one installed while T was active) and so our theory will apply to this implementation just as much as to an implementation that works in a private universe for each SI transaction, and only takes the lock and performs the check and moves the version from private space to the shared database, at the time T commits.

2.3 Multiversion serializability theory

In this paper we will follow the approach of [9], to define the conflict serializability of a multiversion execution. This differs slightly from [4, 12], which define serializability. One difference is that [9] includes extra conflict edges between all writes to an item, whereas [4, 12] don't include edges leading out of a version that is never observed (a situation that can only arise if blind writes occur). Another difference is that [9] deals with a specific version order (that defined by commit times) whereas [4, 12] require only the existance of an order on the versions of an item. For simplicity of the theory, we only consider sequences (total orders) of activity; we do not include in our model the possibility of events that are unordered in time.

A *transaction* T_i is represented as a sequence of actions. Each action is either a read $\mathrm{r}_i[x]$ of some data item x, or else a write $\mathrm{w}_i[y]$ of an item y. The set of items on which T_i has a read action is denoted as *rset*(T_i), and the set of items on which T_i has a write action is *wset*(T_i). We usually omit explicit mention of it, but for precise definitions we consider an initializing transaction $T_{-\infty}$ whch writes the initial version of every item, and a finalizing transaction T_∞ which reads the final version of every item.

For notational simplicity, but without serious loss of generality, this type of model does not allow for the the same item to be accessed in the same way more than once within the transaction; that is, the sequence T_i may not contain two identical actions such as $\ldots \mathrm{w}_i[x] \ldots \mathrm{w}_i[x] \ldots$. As an additional restriction in this paper, we will assume that if T_i contains both $\mathrm{r}_i[x]$ and $\mathrm{w}_i[x]$, then the sequence places $\mathrm{r}_i[x]$ before $\mathrm{w}_i[x]$. This is not a serious limitation on expressiveness, however, since both SI and S2PL mechanisms ensure that when a transaction tries to read an item which it previously wrote, the version observed is the one the transaction itself created; thus the read has no impact on serializability, and it can simply be omitted from the formal model, with the advantage that all the remaining reads in an SI transaction are taken from the transaction's snapshot. To represent an execution of the DBMS, which interleaves the transactions, we use a sequence of operations. The operations which can occur in an execution are the following

- $\mathrm{r}_j[x_k]$, for some transactions T_j and T_k and some item x, representing a read on item x invoked by T_j which returns the value found in the version of x produced by T_k

- $\mathrm{w}_j[x_j]$, for some transaction T_j and some item x, representing a write on item x invoked by T_j which produces a new version

- *start*(T_j) which represents the beginning of execution by transaction T_j

- *commit*(T_j) which represents the commit by transaction T_j

- $\mathrm{slock}_j[x]$ for some transaction T_j and some item x, representing the obtaining of a shared lock on item x on behalf of transaction T_j. (Note that locks are set on items, rather than on versions.)

- $\mathrm{xlock}_j[x]$ for some transaction T_j and some item x, representing the obtaining of an exclusive lock on item x on behalf of transaction T_j

A sequence H of such operations represents an *execution* of the transactions T_1, T_2, ..., T_n, provided the following criteria are all met:

transaction structure For each transaction T_i, there is a translation function h_i that maps each action $\mathrm{w}_i[x]$ of T_i into an operation $\mathrm{w}_i[x_i]$ that occurs in H, and maps each action $\mathrm{r}_i[x]$ of T_i into an operation $\mathrm{r}_i[x_j]$ that occurs in H, for some j such that $w_j[x_j]$ occurs in H.

transaction ordering For any transaction T_i, the operation *start*(T_i) occurs in H before any operation of the form $\mathrm{w}_i[x_i]$, $\mathrm{r}_i[x_k]$ for some k, $\mathrm{slock}_i[x]$ or $\mathrm{xlock}_i[x]$; the operation *commit*(T_i) occurs in H after every operation of the given form; and for any actions o and o' where o is ordered before o' in the transaction T_i, the operation $h_i(o)$ occurs before $h_i(o')$ in H.

lock rules The lock operations are allowed by usual locking rules: whenever $o = \mathrm{slock}_j[x]$ occurs in H, it is preceded by *commit*(T_k) for every transaction T_k such that H contains $\mathrm{xlock}_k[x]$ preceding o; and whenever $o' = \mathrm{xlock}_j[x]$ occurs in H, it is preceded by *commit*(T_k) for every transaction T_k such that H contains either $\mathrm{xlock}_k[x]$ or $\mathrm{slock}_k[x]$ preceding o'.

concurrency control The operations of each transaction follow the concurrency control mechanism allocated for that transaction. If T_i is running with SI, then whenever H contains a write $\mathrm{w}_i[x_i]$, then H contains $\mathrm{xlock}_i[x]$, and whenever H contains $\mathrm{r}_i[x_k]$, then x_k is the version of x whose writer T_k committed most recently before *start*(T_i) in H. On the other hand, if T_i is running with S2PL, then whenever H contains a write $o = \mathrm{w}_i[x_i]$, then H contains $\mathrm{xlock}_i[x]$ before o, and whenever H contains a read $o' = \mathrm{r}_i[x_k]$, then H contains $\mathrm{slock}_i[x]$ before o, and also x_k is the version of x whose writer T_k committed most recently before o' in H.

Note that in our model, we do not include any operations for transactions which do not commit. The assumption here is that an underlying recovery mechanism will rollback any activity of an aborted transaction. Both SI and S2PL ensure that a transaction does not observe any version until the writer of the version has committed.

In order to define the conflict graph for an execution, we first define an order on the versions of a given item. If H contains $\mathrm{w}_j[x_j]$ and $\mathrm{w}_k[x_k]$, then we say that the version x_j precedes the version x_k (written $x_j \ll x_k$) provided *commit*(T_j) occurs before *commit*(T_k) in H. Note that this order is irreflexive, transitive and antisymmetric.

We now define the conflict serialization graph $CSG(H)$. The nodes of $CSG(H)$ are the transactions that occur in H; there are labelled edges which show the implied flow of information (also called *conflict*) which is revealed in operations that conflict, namely two operations which are on versions of the same item, and where at least one version is a write. When we do not need to distinguish between the labels, we write $T_j \rightarrow T_k$ to indicate that at least one of the relations holds: either $T_j \overset{wr}{\rightarrow} T_k$ or $T_j \overset{ww}{\rightarrow} T_k$ or $T_j \overset{rw}{\rightarrow} T_k$.

(wr edge) We say that $T_j \overset{wr}{\underset{x}{\rightarrow}} T_k$ if H contains either an operation $\mathrm{r}_k[x_j]$, or else it contains an operation $\mathrm{r}_k[x_m]$ for a version x_m where $x_j \ll x_m$. We say $T_j \overset{wr}{\rightarrow} T_k$ if there exists some item x such that $T_j \overset{wr}{\underset{x}{\rightarrow}} T_k$.

(ww edge) We say that $T_j \overset{ww}{\vec{x}} T_k$ if H contains two operations $\mathrm{w}_j[x_j]$ and $\mathrm{w}_k[x_k]$, and $x_j \ll x_k$ (recall that this means *commit*(T_j) occurs before *commit*(T_k)). We say $T_j \xrightarrow{ww} T_k$ if there exists x such that $T_j \overset{ww}{\vec{x}} T_k$.

(rw edge) We say that $T_j \overset{rw}{\vec{x}} T_k$ if H contains operations $\mathrm{r}_j[x_m]$ and $\mathrm{w}_k[x_k]$ where $x_m \ll x_k$. Thus T_j reads a version of x and does not see the changes introduced by T_k. We say $T_j \xrightarrow{rw} T_k$ if there exists some item x such that $T_j \overset{rw}{\vec{x}} T_k$. In this case it is traditional to say that there is an *antidependency* from T_j to T_k.

Two executions H and H' on the same set of transactions are called *conflict-equivalent* if $CSG(H) = CSG(H')$. Because an operation $\mathrm{r}_k[x_j]$ is captured by an edge $T_j \overset{rw}{\vec{x}} T_k$, conflict equivalent executions must involve the same read and write operations (in particular, a read must observe the same version in each execution).

An execution is called *1-copy serial* provided all the operations associated to a given transaction T_i occur together in the sequence with no operations of another transaction interleaving, and every read of an item x sees the version of x which was most recently written. This choice of version means that the execution could occur in a system which in fact updated each item in place, and did not allow access to older versions. An execution which is conflict-equivalent to a 1-copy serial execution is called *conflict serializable.*

The main result of serializability theory is that the conflict graph provides a test for conflict serializability.

THEOREM 1. *An execution H is conflict serializable if and only if $CSG(H)$ is acyclic.*

3. CONFLICT THEORY

In this section we use multiversion serializability theory to identify certain conflict patterns which necessarily occur in a non-serializable execution of a set of transactions, some of which are running using SI while the others execute under S2PL. All our statements refer to a particular execution H of a collection of transactions. In the next section, we will consider other "interference" facts, which refer to a set of transactions each with an allocated isolation level, and to the set of all possible executions of those transactions.

We begin this section by examining possible conflict edges formed between two transactions, each of which is running with SI or with S2PL. Next we show a particular pattern of edges that appears in the conflict graph formed in the case where H is an execution that is not conflict-serializable.

For the rest of this section, we will make the universal assumption that H is an execution of a set of transactions T_i for i from 1 to n, in which each transaction is using either S2PL or SI as its concurrency control mechanism (of course, we allow that some transactions are using one while others use the alternative mechanism).

We begin with some simple observations about the possible conflict edges, and the order of operations in H.

LEMMA 1. *If H contains an operation $o = \mathrm{r}_k[x_j]$, then commit(T_j) occurs before $\mathrm{r}_k[x_j]$ in H.*

PROOF. Since H contains $\mathrm{r}_k[x_j]$, by the transaction structure property it must contain an operation $\mathrm{w}_j[x_j]$ which produced the version read by T_k, and by the concurrency control property, that operation $\mathrm{w}_j[x_j]$ must be before o in H.

If T_k is running with SI as its mechanism, then by definition T_j must have been the last writer of x to commit before *start*(T_k); in particular, *commit*(T_j) must occur before *start*(T_k) which is itself before $\mathrm{r}_k[x_j]$.

On the other hand, if T_k is running with S2PL as its mechanism, then it holds a shared lock on x at the time of $\mathrm{r}_k[x_j]$. No matter whether T_j runs with SI or S2PL, it takes an exclusive lock on the item x, before placing the version in public space, and that exclusive lock is held till *commit*(T_j); thus we see that $\mathrm{r}_k[x_j]$ must occur after the exclusive lock is released, that is, after *commit*(T_j).

No matter which concurrency control mechanism is used by T_k, we have shown that *commit*(T_j) occurs in H before $\mathrm{r}_k[x_j]$. $\square$

LEMMA 2. *If commit(T_j) occurs in H before start(T_k) (and therefore every operation of T_j occurs before every operation of T_k), then there does not exist any edge $T_k \rightarrow T_j$ in $CSG(H)$.*

PROOF. We prove the contrapositive. Suppose $T_k \rightarrow T_j$. We must show *commit*(T_j) occurs after *start*(T_k). So consider the three varieties of conflict.

(wr edge) The definition of this conflict shows that H contains either an operation $\mathrm{r}_j[x_k]$, or else it contains an operation $\mathrm{r}_j[x_m]$ for a version x_m where $x_k \ll x_m$. If H contains $\mathrm{r}_j[x_k]$, then by Lemma 1, *commit*(T_k) occurs before $\mathrm{r}_j[x_k]$. On the other hand, if H contains $\mathrm{r}_j[x_m]$ for a version x_m where $x_k \ll x_m$, then again Lemma 1 shows that *commit*(T_m) occurs before $\mathrm{r}_j[x_m]$; however the definition of version order shows *commit*(T_k) is before *commit*(T_m). Thus no matter which operation gives rise to the edge $T_k \xrightarrow{wr} T_j$, we have shown that *commit*(T_k) is before an operation of T_j (either $\mathrm{r}_j[x_k]$ or $\mathrm{r}_j[x_m]$). However the transaction order property shows that *start*(T_k) is before *commit*(T_k) and that any operation of T_j is before *commit*(T_j). Combining these facts shows that *start*(T_k) is before *commit*(T_j), as required.

(ww edge) Here H must contain $\mathrm{w}_k[x_k]$ and $\mathrm{w}_j[x_j]$ for some item x, where $x_k \ll x_j$. The definition of version order shows that *commit*(T_j) must be after *commit*(T_k), but *commit*(T_k) is after *start*(T_k) as H has the transaction order property.

(rw edge) In this case, H contains operations $\mathrm{r}_k[x_m]$ and $\mathrm{w}_j[x_j]$ where $x_m \ll x_j$.

If T_k is using SI as its concurrency control mechanism, then T_m must be the last transaction that writes a version of x and commits before *start*(T_k). Since T_j does write a version of x, and it commits after T_m commits, T_j must not be leigible to form part of the snapshot of T_k. That is, *commit*(T_j) must be after *start*(T_k) as required. On the other hand, if T_k is using S2PL as its concurrency control mechanism, then T_m must be the last transaction that writes a version of x and commits before $\mathrm{r}_k(x_m)$. Since T_j does write a version of x, and it commits after T_m commits, T_j must not be eligible to be seen by the read. That is, *commit*(T_j) must be after $\mathrm{r}_k[x_m]$, which is itself after *start*(T_k) by the transaction order property. Thus no matter which concurrency control mechanism is used by T_k, we have shown *commit*(T_j) must be after *start*(T_k), as required.

For each variety of conflict edge we have shown the result. $\square$

We now give the key technical results which will drive the theory we develop.

LEMMA 3. *If $CSG(H)$ contains a conflict edge $T_j \rightarrow T_k$, and if commit(T_k) occurs before commit(T_j) in H, then*

(i) *rset*$(T_j) \cap$ *wset*$(T_k) \neq \emptyset$

(ii) *wset*$(T_j) \cap$ *wset*$(T_k) = \emptyset$

(iii) T_j *is running at SI in H*

PROOF. We use case analysis on the different types of conflict in the definition of $T_j \rightarrow T_k$.

- $T_j \overset{ww}{\rightarrow} T_k$.
  Here T_j writes a version, say version x_j of item x, and T_k writes a later version x_k, where $x_j \ll x_k$. By the definition of version order, *commit*(T_k) is after *commit*(T_j) in H. This contradicts the hypothesis of this lemma; thus this case can't occur.

- $T_j \overset{wr}{\rightarrow} T_k$.
  Here T_j writes a version, say version x_j of item x, and T_k reads that version, or else T_k reads a later version. That is, H contains $\mathrm{r}_k[x_j]$ or else H contains $\mathrm{r}_k[x_m]$ for some m where $x_j \ll x_m$. If H contains $\mathrm{r}_k[x_j]$, then by Lemma 1, in H the operation *commit*(T_j) occurs before $\mathrm{r}_k[x_j]$, which itself occurs before *commit*(T_k). On the other hand, if H contains $\mathrm{r}_k[x_m]$ for $x_j \ll x_m$, then by Lemma 1, in H the operation *commit*(T_m) occurs before $\mathrm{r}_k[x_m]$; however, by the definition of version ordering *commit*(T_j) occurs before *commit*(T_m), and by the nature of transactions $\mathrm{r}_k[x_m]$ occurs before *commit*(T_k). Thus we see that again *commit*(T_j) occurs in H before *commit*(T_k). This contradicts the hypothesis of this lemma; and so this case can't occur.

- $T_j \overset{rw}{\rightarrow} T_k$.
  Here H contains an operation $\mathrm{r}_j[x_n]$ and an operation $\mathrm{w}_k[x_k]$, and $x_n \ll x_k$. That is, T_j reads an item x and doesn't see the version written by T_k. This already proves (i), since $x \in$ *rset*$(T_j) \cap$ *wset*(T_k).

  If T_j is using S2PL as its concurrency control mechanism, then it will keep the corresponding shared lock on x till *commit*(T_j), so installation of version x_k will be later, after the shared lock is released, because T_k must have an exclusive lock when its version is installed (this is so whether T_k uses S2PL or SI). Since T_k's version is installed at *commit*(T_k), this contradicts the hypothesis that *commit*(T_k) precedes *commit*(T_j). Thus we have proved (iii): that T_j is running under SI.

  Now because an SI transaction works on a snapshot containing the committed state when the reader starts, and because T_j does not observe the version written by T_k, we must have that *commit*(T_k) is after *start*(T_j) in H; combined with the hypothesis that *commit*(T_k) is before *commit*(T_j) this says that *commit*(T_k) is during the active period of T_j. Now if there is any item in intersecting writesets, then the first committer wins rule of SI will abort T_j. This contradicts the assumption that T_j commits! Thus there is no item in the intersection of the writesets of T_j and T_k, proving (ii).

For each variety of conflict edge we have shown the result. $\square$

LEMMA 4. *If in $CSG(H)$ there is a conflict edge $T_j \rightarrow T_k$ such that commit(T_j) occurs after start(T_k) in H, and T_k is running at SI, then*

(i) *rset*$(T_j) \cap$ *wset*$(T_k) \neq \emptyset$

(ii) *wset*$(T_j) \cap$ *wset*$(T_k) = \emptyset$

PROOF. We use case analysis on the different types of conflict in the definition.

- $T_j \overset{ww}{\rightarrow} T_k$.
  T_j writes a version x_j of some item x and T_k writes a later version x_k, so by definition of version order, *commit*(T_k) occurs in H after *commit*(T_j). Combined with the hypothesis of this lemma, we have that *commit*(T_j) occurs while T_k is running. That is, a version of x was installed by T_j while T_k was running, and T_k also installed a version of the same item. This contradicts the First Committer Wins rule for T_k which we recall is hypothesized to be using SI as its concurrency control. Thus this case can't happen.

- $T_j \overset{wr}{\rightarrow} T_k$.
  Here T_j writes a version, say version x_j of item x, and T_k reads that version, or else T_k reads a later version. That is, H contains $\mathrm{r}_k[x_j]$ or else H contains $\mathrm{r}_k[x_m]$ for some m where $x_j \ll x_m$. Because T_k is using SI, the version produced by T_j (or a later version) forms part of T_k's snapshot, which contradicts the hypothesis that *commit*(T_j) is after *start*(T_k). Thus this case can't happen.

- $T_j \overset{rw}{\rightarrow} T_k$.
  Here H contains an operation $\mathrm{r}_j[x_n]$ and an operation $\mathrm{w}_k[x_k]$, and $x_n \ll x_k$. That is, T_j reads an item x and doesn't see the version written by T_k. This already proves (i) since $x \in$ *rset*$(T_j) \cap$ *wset*(T_k). To prove (ii) we divide the discussion based on the concurrency control mechanism used by T_j.

  – If T_j is using S2PL as its mechanism, then it will keep the shared lock on x obtained for $\mathrm{r}_j[x_n]$ till *commit*(T_j), so installation of the version x_k (which by the nature of T_k's SI mechanism happens at *commit*(T_k) when T_k has an exclusive lock on x) must be later. That is, *commit*(T_j) occurs during execution of T_k. If there was any item in intersection of writesets, the First committer wins rule of SI transactions would abort T_k. Thus in this case there is no item in intersection of writesets.

  – If T_j is using SI, then either *commit*(T_j) is before *commit*(T_k) or vice versa. In the first case, *commit*(T_j) is during execution of T_k, so if there are any items in intersecting writesets, then first committer wins rule of SI would abort T_k. Thus in this case the intersection of writesets must be empty. On the other hand, if *commit*(T_k) is before *commit*(T_j), then since T_j doesn't see T_k in it's snapshot, we must have that *commit*(T_k) occurs during execution of the SI transaction T_j. Again, first committer wins implies that there is no item in intersection of writesets.

In both cases we have proved (ii).

For each variety of conflict edge we have shown the result. $\square$

Based on these lemmas, we can state and prove the key characterization of cycles in the conflict graph. This strengthens a result of [1] which showed that when all transactions use SI, there are two consecutive anti-dependency edges in a cycle. The importance of this theorem will appear in the next section, where we will identify as "pivots" those transactions which might fill the role of T_B in the following.

Theorem 2. *If H is not conflict-serializable, then the conflict serialization graph $CSG(H)$ contains a chord-free cycle $T_\alpha \to T_\beta \to \ldots T_\eta \to T_\alpha$, within which there are three consecutive transactions which we will call T_A, T_B and T_C (here T_C is allowed to equal T_A, if the cycle has length 2) such that*

(i) *T_B runs with SI as its concurrency control mechanism*

(ii) *rset*$(T_B) \cap$ *wset*$(T_C) \neq \emptyset$ *and wset*$(T_B) \cap$ *wset*$(T_C) = \emptyset$

(iii) *rset*$(T_A) \cap$ *wset*$(T_B) \neq \emptyset$ *and wset*$(T_A) \cap$ *wset*$(T_B) = \emptyset$

Proof. Since H is a multiversion history which is not conflict-serializable, there must be at least one cycle in $CSG(H)$. Therefore, there is a cycle of minimal length, which we call $T_\alpha, T_\beta \ldots T_\eta, T_\alpha$. Since this cycle has minimal length, it must be chord-free. Choose T_C to be the earliest to commit among the transactions $T_\alpha \ldots T_\eta$ that make up the cycle, and then choose T_A and T_B to be its two immediate predecessors in the cycle. By choice of T_C, *commit*(T_B) is after *commit*(T_C) so we apply Lemma 3 to conclude (i) and (ii). We also note that *start*(T_B) must be before *commit*(T_C), as otherwise Lemma 2 shows that T_B could not have any conflict to T_C. But, by choice of T_C, *commit*(T_A) is after *commit*(T_C), hence we conclude that *commit*(T_A) is after *start*(T_B), so we apply Lemma 4 and (i) to conclude (iii). $\square$

4. GLOBAL INTERFERENCE THEORY

In this section we consider a given set of transactions $\mathcal{T} = T_1, T_2, \ldots, T_n$ and a given *allocation* which chooses, for each transaction, an appropriate concurrency control mechanism (either SI or S2PL). We will identify an allocation by stating which subset $\mathcal{S}$ of transactions are allocated to run with SI as their mechanism (this of course indicates that the transactions in the complement $\mathcal{T} - \mathcal{S}$ are each allocated to run with S2PL as their mechanism). We say that the allocation $\mathcal{S}$ for the set of transactions $\mathcal{T}$ is *acceptable* if every execution H that can arise is conflict-serializable.

To characterize when an allocation for a set of transactions is acceptable, we will produce a graph showing which transactions *interfere with* which other transactions. This will give a global[3] directed graph called the interference graph $IG(\mathcal{T})$, with two varieties of edges. Certain nodes in the graph will be distinguished as *pivots*, and our main result will be that the allocation is acceptable (as defined in the previous paragraph) exactly when the pivots are all allocated to use S2PL as their concurrency control mechanism.

The *interference graph*, denoted as $IG(\mathcal{T})$, has the transactions T_1, T_2, ..., T_n as nodes, and contains edges which may be *exposed* (shown $T_j \xrightarrow{expo} T_k$) or *protected* (shown as $T_j \xrightarrow{prot} T_k$). The definition of these concepts is inspired by the conditions (ii) and (iii) in Theorem 2. When we do not need to distinguish between varieties of edges, we use the notation $T_j \xrightarrow{glob} T_k$ to indicate some interference edge exists.

For any ordered pair of transactions we have one of the following three situations.

(no edge) There is no interference edge from T_j to T_k when all the following occur: *rset*$(T_j) \cap$ *wset*$(T_k) = \emptyset$, *wset*$(T_j) \cap$ *rset*$(T_k) = \emptyset$, and *wset*$(T_j) \cap$ *wset*$(T_k) = \emptyset$. Thus the only items accessed in both transactions are not modified in either.

(exposed edge) There is an exposed edge $T_j \xrightarrow{expo} T_k$ when both *rset*$(T_j) \cap$ *wset*$(T_k) \neq \emptyset$, and also *wset*$(T_j) \cap$ *wset*$(T_k) = \emptyset$.

(protected edge) There is an protected edge $T_j \xrightarrow{prot} T_k$ when one of two situations is found

- *wset*$(T_j) \cap$ *wset*$(T_k) \neq \emptyset$, or
- (*rset*$(T_j) \cap$ *wset*$(T_k) = \emptyset$, and *wset*$(T_j) \cap$ *rset*$(T_k) \neq \emptyset$).

We note that these cases are a partition: for any ordered pair T_j and T_k, there will either be no edge, or exactly one variety of interference edge, from T_j to T_k. Furthermore, we note that the interference edges come in pairs: when there is an edge from T_j to T_k then there is also an edge in the reverse direction, from T_k to T_j. However, in such a pair, the variety of edge in each direction can be the same, or it can be different.

It is important to distinguish between the interference edges of $IG(\mathcal{T})$, which are written with long arrows and indicate the potential for interference between the transactions due to common access to some item, compared to the short arrow conflict edges of a dynamic conflict graph $CSG(H)$ for a particular execution H. A relationship between these is shown next, showing how the dynamic conflict graph is a subgraph of the intereference graph, containing at least one of the directions for each pair or reversed interference edges.

Lemma 5. *If $T_j \to T_k$ in $CSG(H)$ then $T_j \xrightarrow{glob} T_k$ in $IG(\mathcal{T})$. As a partial converse, if $T_j \xrightarrow{glob} T_k$ in $IG(\mathcal{T})$, then either $T_j \to T_k$ in $CSG(H)$ or else $T_j \to T_k$ in $CSG(H)$.*

Proof. Suppose $T_j \to T_k$. We proceed by cases, depending on the variety of conflict. If $T_j \overset{ww}{\underset{x}{\to}} T_k$, then H contains both $w_j[x_j]$ and $w_k[x_k]$. By the transaction structure

[3]Global here indicates that the graph is determined by the transactions and allocation, and not by any particular execution H. In fact, the definition we give will be independent of the allocation, and determined only by the transactions.

property of an execution, this means that T_j contains $w_j[x]$, and T_k contains $w_k[x]$. Thus $x \in wset(T_j) \cap wset(T_k)$, so $T_j \xrightarrow{glob} T_k$. If $T_j \overset{wr}{\vec{x}} T_k$, then H contains either $r_k[x_j]$ or else $r_k[x_m]$ where $x_j \ll x_m$. Whichever is present, transaction structure shows that T_k contains $r_k[x]$. Also, we deduce in either case the existance of $w_j[x_j]$ and so T_j contains $w_j[x]$. Thus $x \in (wset(T_j) \cap rset(T_k))$, and so $T_j \xrightarrow{glob} T_k$. Similarly if $T_j \overset{rw}{\vec{x}} T_k$, then $x \in (rset(T_j) \cap wset(T_k))$, so $T_j \xrightarrow{glob} T_k$.

For the partial converse, suppose $T_j \xrightarrow{glob} T_k$ in $IG(\mathcal{T})$. Thus there is some item x in one of the intersections $rset(T_j) \cap wset(T_k)$, $wset(T_j) \cap rset(T_k)$, or $wset(T_j) \cap wset(T_k)$.

Suppose $x \in wset(T_j) \cap wset(T_k)$. Then by transaction structure, H contains both $w_j[x_j]$ and $w_k[x_k]$. Depending on which transaction commits first, either $x_j \ll x_k$ or else $x_k \ll x_j$. In the first case, $T_j \overset{ww}{\to} T_k$ (and so $T_j \to T_k$), while in the second case $T_k \overset{ww}{\to} T_j$ (and so $T_k \to T_j$). On the other hand, suppose $x \in rset(T_j) \cap wset(T_k)$; then H contains $r_j[x_m]$ for some m, and also H contains $w_k[x_k]$. One of the three following cases must hold: either $T_m = T_k$ (in which case $T_k \overset{wr}{\to} T_j$) or else T_m commits after T_k (in which case also $T_k \overset{wr}{\to} T_j$) or else T_m commits before T_k (in which case $T_j \overset{rw}{\to} T_k$). Finally, suppose $x \in wset(T_j) \cap rset(T_k)$; by the same argument, we show either $T_j \overset{wr}{\vec{x}} T_k$ or else $T_k \overset{rw}{\vec{x}} T_j$. □

We say that a node T_B in $IG(\mathcal{T})$ is a *pivot* if there exist transactions T_A and T_C (which may be equal to each other), such that the following three properties all hold

- $T_A \xrightarrow{expo} T_B$
- $T_B \xrightarrow{expo} T_C$
- T_A, T_B and T_C occur consecutively in a chord-free cycle $T_\alpha \xrightarrow{glob} T_\beta \xrightarrow{glob} \ldots T_\eta \xrightarrow{glob} T_\alpha$ in $IG(\mathcal{T})$.

As a special case, note that whenever there are reversed exposed edges $T_j \xrightarrow{expo} T_k$ and $T_k \xrightarrow{expo} T_j$, then the definition declares both T_j and T_k to be pivots, in a chord-free cycle of length 2 where both edges are exposed.

To further illustrate the definition of pivot, consider the following transactions in a database with data items u, v, x, y, and z.

- $T_1 = r_1[x]r_1[y]w_1[x]$
- $T_2 = r_2[v]r_2[y]r_2[z]w_2[v]w_2[y]$
- $T_3 = r_3[u]r_3[z]w_3[u]w_3[z]$
- $T_4 = r_4[u]r_4[v]r_4[x]w_4[u]w_4[v]$

The graph $IG(T_1, T_2, T_3, T_4)$ is shown in Figure 1. For easier visualization, we draw exposed edges with dashed lines, while protected edges are solid. Notice that the edge from T_4 to T_1 is exposed because x is in $rset(T_4) \cap wset(T_1)$, while $wset(T_4) \cap wset(T_1)$ is empty. The edge from T_1 to T_4 is protected because $(rset(T_1) \cap wset(T_4) = \emptyset$, and also $x \in wset(T_1) \cap rset(T_4)$. The edge from T_3 to T_4 is protected because $u \in wset(T_3) \cap wset(T_4)$.

In this example, T_1 is a pivot, because of the chord-free cycle $T_4 \xrightarrow{expo} T_1 \xrightarrow{expo} T_2 \xrightarrow{glob} T_4$. The other nodes are not pivots. T_3 is not a pivot because it has no outgoing exposed

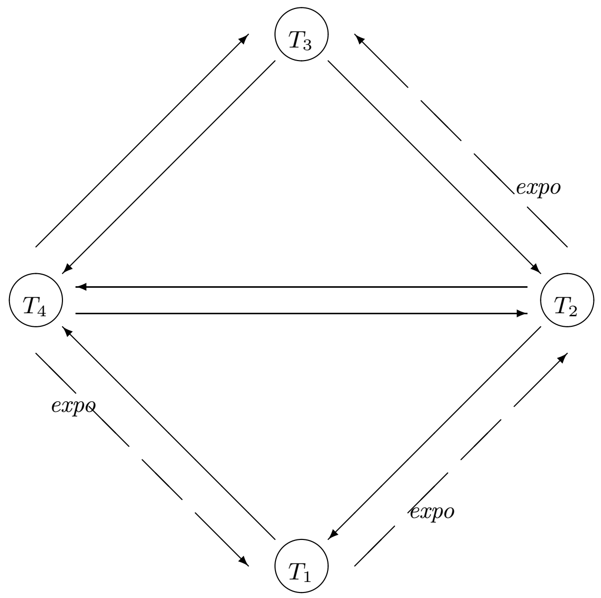

Figure 1: Example Interference Graph

edge; T_4 is not a pivot because it has no incoming exposed edge; and T_2 is not a pivot because its incoming and outgoing exposed edges (namely $T_1 \xrightarrow{expo} T_2 \xrightarrow{expo} T_3$) can't be joined in a chord-free cycle (the only walks from T_3 back to T_1 go either though T_2 which would give a repeated vertex, or through T_4 where there is a chord directly from T_4 to T_2).

We now present our main theorem, which shows exactly when the allocation $\mathcal{S} \subset \mathcal{T}$ is acceptable.

Theorem 3. *For a set of transactions $\mathcal{T}$, an allocation $\mathcal{S}$ is acceptable if and only if none of the pivots of $IG(\mathcal{T})$ is in $\mathcal{S}$.*

Proof. We actually prove the contrapositive statement: there exists an execution which is not conflict-serializable, if and only if some pivot T_B is in $\mathcal{S}$ and so is allocated to run using SI as its mechanism.

So suppose that there is an execution H which is not conflict-serializable. By Theorem 2, there is in $CSG(H)$ a chord-free cycle $T_\alpha \to T_\beta \to \ldots T_\eta \to T_\alpha$, within which there are three consecutive transactions T_A, T_B and T_C such that (i) T_B runs with SI as its concurrency control mechanism; (ii) $T_B \xrightarrow{expo} T_C$ in $IC(\mathcal{T})$ (iii) $T_A \xrightarrow{expo} T_B$ in $IC(\mathcal{T})$. We note that Lemma 5 immediately allows us to find a corresponding cycle $T_\alpha \xrightarrow{glob} T_\beta \xrightarrow{glob} \ldots T_\eta \xrightarrow{glob} T_\alpha$ in $IC(\mathcal{T})$, and we claim that this cycle is also chord-free. Once we have proved the claim, we are done, as this chord-free cycle in $IC(\mathcal{T})$ shows that T_B is a pivot, and by (i) above T_B runs with SI.

Thus we must only prove the claim that the cycle is chord-free. Suppose for the sake of contradiction that some chord existed in $IC(\mathcal{T})$, say $T_\phi \xrightarrow{glob} T_\psi$. By Lemma 5, either $T_\phi \to T_\psi$ in $CSG(H)$, or else $T_\psi \to T_\phi$. In either case, the appropriate edge forms a chord in the cycle in $CSG(H)$, contradicting our choice as a chord-free cycle in $CSG(H)$.

On the other hand, suppose that there is a transaction T_B which is a pivot and is in S. By definition of a pivot, there is a chord-free cycle in $IG(\mathcal{T})$ through T_B where the edges into and out of T_B are exposed. That is, (since the cycle can be written so it starts and ends at T_B) we can identify $T_B \xrightarrow{expo} T_\beta \xrightarrow{glob} T_\gamma \ldots T_\eta \xrightarrow{expo} T_B$. Consider the following sequence of operations H in which every transaction except T_B runs serially, with all its operations together. H has $start(T_B)$ as its first operation, followed by all the operations[4] of T_β, then all the operations of T_γ, then all the operations of each other transaction from the chord-free cycle (in order as the transactions appear in the cycle), then all the operations of T_B except for its start, then all the operations of each other transaction that did not appear in the chord-free cycle (in an arbitrary order of transactions).

We claim that H is an execution which can occur for the given transactions and allocation. To prove the claim, notice first that whenever transaction T_i is performing reads and writes in H, then no other transaction can hold any locks (because the only other transaction that might be running is T_B, and if so, it has performed no reads or writes).

So if T_i is using S2PL as its mechanism, its operations can proceed without delay. On the other hand, if T_i is not equal to T_B and it is using SI as its mechanism, its reads can of course proceed (since reads are never delayed in SI), and its versions can be produced and take the exclusive locks they need ready to be installed; furthermore, the First Committer Wins rule will not prevent its commit, since no other transaction has committed while T_i was active. Finally, for T_B itself (which is using SI as its mechanism), its reads can proceed, and its versions can be produced and obtain the exclusive locks so they are ready to be installed; also T_B will not be aborted by the First Committer wins rule, since none of the transactions that committed during T_B's activity (namely T_β, T_γ, … T_η) modify any item that T_B wrote. This last statement is justified by the fact that T_β and T_η have an exposed edge with T_B (and so have empty intersection of write sets), while the other nodes in the cycle have no interference edge at all with T_B (because the cycle is chord-free), and so they also have empty intersection of writesets with T_B.

Furthermore, H is not conflict serializable, since we claim that the dynamic conflict graph $CSG(H)$ contains a cycle $T_B \rightarrow T_\beta \rightarrow T_\gamma \ldots T_\eta \rightarrow T_B$. Here is the proof of the claim.

- For each edge in the cycle in $IG(\mathcal{T})$ which does not involve T_B, say $T_\gamma \xrightarrow{glob} T_\delta$, Lemma 5 shows that either $T_\gamma \rightarrow T_\delta$ in $CSG(H)$ or else $T_\delta \rightarrow T_\gamma$; however the latter is impossible by Lemma 2 since in H T_γ has committed before any operation of T_δ occurs. Thus we must have $T_\gamma \rightarrow T_\delta$ in $CSG(H)$.

- By choice of T_B we have $T_B \xrightarrow{expo} T_\beta$ in $IG(\mathcal{T})$, that is, there is some item x in $rset(T_B) \cap wset(T_\beta)$. By construction, $start(T_B)$ occurs in H before any transaction has committed, so the snapshot for T_B (which we recall is using SI as its mechanism) is the initial state. Thus when T_B reads x, it will see the initial version, and not see the version written by T_β. That is, there is $T_B \overset{rw}{\rightarrow} T_\beta$ in $CSG(H)$.

- By choice of T_B we have $T_\eta \xrightarrow{expo} T_B$ in $IG(\mathcal{T})$, that is there is some item y in $rset(T_\eta) \cap wset(T_B)$. By construction all operations of T_η occur in H before $\mathrm{w}_B[y_B]$, and so whichever version of y is read by T_η is not the version which will later be produced by T_B. That is, there is $T_\eta \overset{rw}{\rightarrow} T_B$ in $CSG(H)$.

[4]The operations of T_β are $start(T_\beta)$, followed by all the read and write actions of T_β under appropriate translation function h_β (interspersed with needed locking operations of T_β), and concluded by $commit(T_\beta)$.

This completes the demonstration of a cycle in $CSG(H)$, showing that H is not conflict serializable. $\square$

For the set of transactions shown in Figure 1, we see that any allocation is acceptable as long as T_1 is allocated to use S2PL as its concurrency control mechanism; the other transactions T_2, T_3, and T_4 can be allocated independently and arbitrarily to use either SI or S2PL.

5. THE LATTICE OF ALLOCATIONS

In this section, we raise the level of abstraction, by thinking not of a particular allocation, but rather we look at the structure of the space $\mathcal{A}$ consisting of all possible allocations. Clearly, we can put a partial order on $\mathcal{A}$. If a_1 and a_2 are allocations, then we say $a_1 \preceq a_2$ if for every transaction T_i, a_1 allocates T_i to an equal or weaker (more permissive) isolation mechanism than a_2 allocates for T_i.

Recall that we have identified each allocation with the set of transactions which run with SI as their concurrency control mechanism. So $\mathcal{S}_1 \preceq \mathcal{S}_2$ means $\mathcal{S}_1 \subseteq \mathcal{S}_2$; that is, the lesser allocation has fewer transactions using SI. Thus the bottom element of this lattice is the allocation where all transactions use S2PL; this is certainly acceptable. The top element of $\mathcal{A}$ is the allocation where all transactions use SI; this may or may not be acceptable, depending on the nature of the transactions.

Now consider the subset of acceptable allocations, denoted by $\mathcal{A}^{SR}$. Theorem 3 can be expressed as $\mathcal{A}^{SR} = \{\mathcal{S} : \mathcal{S} \preceq (\mathcal{T} - \mathcal{P})\}$ where $\mathcal{P}$ denotes the set of pivots in $IG(\mathcal{T})$. In the language of lattice theory, $\mathcal{A}^{SR}$ is a principal ideal in $\mathcal{A}$.

This has two consequences. The first says that if we are given an acceptable allocation, and change some transactions from using SI to using S2PL, then we still have an acceptable allocation. That is, increasing the strength of each transaction's allocated concurrency mechanism can't introduce non-serializable executions. This is obvious, and it could be proved directly, without using any of our new results.

Corollary 1. *If $\mathcal{S}_1$ is an acceptable allocation, and $\mathcal{S}_2 \subseteq \mathcal{S}_1$, then $\mathcal{S}_2$ is acceptable.*

The second consequence is much more surprising. It says that the join of acceptable allocations (which uses for each transaction whichever mechanism is the weaker, among those used in the given allocations) is itself acceptable. We are unaware of any way to prove this directly.

Corollary 2. *If $\mathcal{S}_1$ and $\mathcal{S}_2$ are both acceptable allocations, then the allocation $\mathcal{S}_1 \cup \mathcal{S}_2$ is also acceptable.*

6. CONCLUSIONS AND FURTHER WORK

We have produced the first body of theory which allows determining the serializability of all executions in a system where different concurrency control mechanisms are used for different transactions. We have also been able to characterize exactly which transactions need to be allocated to use S2PL, and which can be allowed to use either SI or S2PL These results can be used when running transactions in a

DBMS such as the forthcoming Yukon release of Microsoft SQL Server, which offers both concurrency control mechanisms; they can also be used when running on a platform such as Oracle, where SI is provided but the application can obtain S2PL by setting (and holding) explicit shared locks before reading.

In future work, we intend to study the performance implications of the different acceptable allocations. Intuition would suggest the maximal acceptable allocation (using S2PL as little as possible, and SI as much as possible) would give the best throughput; however this needs to be carefully checked. We also will seek to incorporate our results into tools that will examine a mix of application programs (which are not explicit sequences of reads and writes, but rather contain SQL statements and control flow) in order to determine suitable choices of isolation level for each program. In a more theoretical direction, we hope to extend our work to characterize acceptable allocations where some transactions use the Read Committed isolation level (that is, they take short duration shared locks before reading) while others use SI and still others use S2PL.

7. ACKNOWLEDGMENTS

The author thanks the University of Washington in Seattle, which hosted the sabbatical during which the first stages of this work occurred. He is also very grateful for constructive feedback on these ideas from members of the Database Research Group at Microsoft Research, especially from Phil Bernstein. The presentation was greatly improved by comments from Uwe Roehm and Shirley Goldrei.

8. REFERENCES

[1] A. Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions (PhD thesis). Technical Report TR-786, Laboratory for Computer Science, Massachusetts Institute of Technology, March 1999.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 1–10. ACM, June 1995.

[3] A. Bernstein, P. Lewis, and S. Lu. Semantic conditions for correctness at different isolation levels. In *Proceedings of IEEE International Conference on Data Engineering*, pages 57–66. IEEE, February 2000.

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

[5] A. Fekete. Serializability and snapshot isolation. In *Proceedings of Australian Database Conference*, pages 201–210. Australian Computer Society, January 1999.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, California, 1993.

[7] K. Jacobs. Concurrency control: Transaction isolation and serializability in SQL92 and Oracle7. Technical report, Oracle White Paper, Part No A33745, July 1995.

[8] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[9] Y. Raz. Commitment ordering based distributed concurrency control for bridging single and multi version resources. In *Proceedings of Third International Workshop or Research Issues in Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS)*, pages 189–198. IEEE, June 1993.

[10] R. Schenkel and G. Weikum. Integrating snapshot isolation into transactional federations. In *Proceedings of 5th IFCIS International Conference on Cooperative Information Systems (CoopIS 2000)*, pages 90–101, September 2000.

[11] W. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Trans. Program. Lang. Syst.*, 11(2):249–283, April 1989.

[12] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, San Francisco, California, 2002.