Lower Isolation Levels in Distributed Transactions

Tim Kempster, Colin Stirling, Peter Thanisch
Department of Computer Science, University of Edinburgh

September 14, 1999

Abstract

We extend the notion of conflicting actions within transaction schedules and derive a definition of conflict serializability. Phenomenon based definitions of isolation have been presented in[1][2]. We discuss these and derive similar definitions for distributed schedules. We model a simple distributed database and show how the lower level of transaction isolation READ COMMITTED can be raised to full transaction isolation by passing extra information in the messages of a distributed commit protocol.

1 Introduction

A recent trend in the search for transaction processing performance improvements has been the exploitation of concurrency opportunities that occur when transactions request a lower level of isolation, i.e. the 'I' in the ACID properties of transactions. Different levels of isolation are now a part of the ANSI SQL standard[1] and many DBMS vendors have implemented this facility, or something similar.

The highest level of isolation sometimes referred to as full isolation is serializability. In this paper we provide a characterisation of conflict serializability that includes the abort or commit outcome of transactions. Within this framework we define full or serializable transaction isolation. Much of the previous work in this area has relied on assumptions about transaction recoverability[4] we do not require these assumptions here.

In order to avoid reference to a particular concurrency control technique (e.g. locking) isolation levels have been defined by the types of isolation *phenomena* that can be found in schedules[1][2][9]. In this paper we include a phenomenon based definition of transaction isolation for centralised database schedules.

In a distributed database data items reside at different sites in the systems. In order to give an account of conflict serializability in distributed databases we define distributed schedules as a vector of centralised or local schedules (one for each site in the system). In a distributed schedule transaction must be atomic, the commit or abort outcomes of a transactions must agree at all sites in the system. A commit protocol, the most popular of this being two-phase commit, is used to ensure this. We argue that all distributed schedules must exhibit a property we call causal commitment. We then show that excluding the same phenomena we defined for centralised schedules from each of the local schedules that constitute a distributed schedule together with the property of causal commitment ensures that distributed schedules are serializable.

We discuss the use of locking to exclude phenomena from schedules and thus ensure serializability. A level of isolation deemed READ COMMITTED by the ANSI community is examined and we discuss the level of locking that must be achieved to ensure this. This reduced level of transaction isolation produces greater transaction concurrency but allows non-serializable schedules. It is possible however to detect when the level of isolation has been compromised and abort transactions to regain serializability. This combination of locking with validation at commit time was discussed in[4].

Lastly, we model a simple distributed database. In this model we use a level of locking to provide the lower READ COMMITTED isolation level, but show by exchanging information about conflicts during the commit protocol how this level of isolation can be raised to full or serializable isolation.

Much of the work in this paper has been discussed elsewhere but we believe that our account consolidates many of the past results and provides a good framework in which to understand issues of distributed transaction isolation.

2 Centralised Schedules

2.1 Transactions and Schedules

Let t_i, t_j denote transactions and d, d' denote data items in a database. It is assumed that $t_i \neq t_j$, unless otherwise stated, but we do not assume $d \neq d'$. A transaction t_i carries out actions of four different types: read and write *accesses* to a data item, which we denote $r_i[d]$, $w_i[d]$ respectively or $o_i[d]$ to denote either a $r_i[d]$ or $w_i[d]$, and commit and abort actions which we call *terminals* and denote c_i, a_i respectively or e_i to denote either c_i or a_i. By the time a transaction terminates either, in the case of commit, all the changes it has made, or in the case of abort, all the changes are undone. Furthermore these changes or undo operations are made in a *durable* fashion so that they will survive system failure. Once the transaction has committed, the output of the transaction is returned to the user [1]. If a transaction aborts, all write actions for that transaction are undone and either the abort outcome is conveyed to the user or the transaction is restarted at a later time.

Accesses		Terminals	
$w_i[d]$	t_i writes d	c_i	t_i commits
$r_i[d]$	t_i reads d	a_i	t_i aborts

We assume each type of access within a transaction is to a unique data item[2].

To increase transaction throughput, transactions are executed concurrently. An order on the actions of the concurrently executing transactions is dictated by a scheduler. We call the sequence[3] of actions produced by the scheduler a *schedule*, and denote this, σ. An example of a schedule generated when t_1 and t_2 execute concurrently can be written as

$$r_1[d]\, r_2[d]\, w_1[d']\, w_2[d]\, c_1[s]\, c_2[s]$$

We use the convention that if one action is to the right of another the rightmost action, of the two actions, took place after the leftmost in

[1] In some transaction processing applications there is an interactive dialogue between the user the transaction. Any results passed to the user are not confirmed until the user is told that the transaction has committed.

[2] The results in this paper do not depend on this but it is a useful notational convenience. Disallowing repeated accesses to the same data item does not prevent isolation anomalies.

[3] In fact, a partial order is all that is required, but to simplify notation we consider only the case of a total order in this paper.

the schedule. In any schedule no transaction access will be scheduled after its terminal. A *serial* schedule is one in which each transaction executes all its actions before the next. A *complete schedule* is one in which all transactions in the schedule have a terminal in that schedule. If, in a schedule σ, a transaction action $o_i[d]$ exists but no terminal e_i exists the schedule is a *prefix* of a complete schedule. Any prefix σ_{pre} can be transformed into a complete schedule σ_{com} by forming the abort completion.

Definition 1 The *abort completion* of an incomplete schedule σ_{inc} is formed by appending an a_i to σ_{inc} for each transaction t_i that has an access $o_i[d]$ but no terminal e_i in σ_{inc}.

□

An important property of schedules is transaction *isolation*. A scheduler maintains a level of isolation by imposing constraints on the allowable interleaving of transactions' actions within the schedule. We will now proceed to define the strictest isolation level *serializable isolation* often referred to as *serializability*[4].

2.2 Conflicts and Conflict Equivalence

In order to make assertions about properties of schedules we develop some notation. We say $o_i[d] \prec o'_j[d]$ if an action $o_i[d]$ is ordered before action $o'_j[d]$ in σ. We say $c_i[s]$ $(a_i[s])$ is true in a schedule if action $c_i[s]$ $(a_i[s])$ is in the schedule. We use $w_i[d] \prec c_j$, to denote that t_j commits and does so after a write action of t_i on data item d. We write $w_i[d] \prec e_j$, to mean that either t_j aborts or commits, and does so after a write action by t_i on data item d.

Similarly, we write $r_j[d] \prec (a_i \wedge c_j)$, to say t_i aborts after a read of d by t_j and also t_j commits. Finally, we write $(r_i[d] \prec w_j[d]) \wedge a_i \wedge c_j$, to say t_i aborts and t_j commits, and that $r_i[d]$ is before $w_j[d]$, this allows a_i before or after w_j.

To define serializability of schedules we must first define equivalence between schedules. The most common and useful definition is that of conflict equivalence [4].

Two transactions t_i and t_j are said to *conflict* if both transactions access the same data item d and if transposing the order of these

[4]The ANSI standard [1] calls this level of isolation REPEATABLE READ, they reserve the term SERIALIZABILITY for a stronger form of isolation excluding phantoms.

accesses on d *might* result in either a different value being read by one of the transactions, or a different value resulting at d after the transactions complete.

For example, if t_1 reads d before t_2 writes d and then both t_1 and t_2 commit, as in the schedule above, t_1 is ordered before t_2 by their conflict on d. To see this note t_1 might read a different value if its read were after t_2's write. Using the notation we developed earlier we can list all the possible types of conflict that can occur in a schedule.

I $r_i[d] \prec w_j[d] \wedge (c_i \wedge c_j)$

II $w_i[d] \prec r_j[d] \wedge (c_i \wedge c_j)$

III $w_i[d] \prec w_j[d] \wedge (c_i \wedge c_j)$

IV $r_i[d] \prec w_j[d] \wedge (c_i \wedge a_j)$

V $w_i[d] \prec r_j[d] \prec (a_i \wedge c_j)$

In much of the literature on the classical theory of serializability there is an assumption that a mechanism exists to ensure schedules are recoverable, and often that they avoid the possibility of cascading aborts[5]. Under this assumption the classical theory need not distinguish between two schedules, say s_1 and s_2, where the only difference is that, in s_1 a transaction, t_1, read a value for a data item that was written by another transaction, t_2, *before* t_2 aborted because such schedules are disallowed by recoverability constraints.

$$s_1 : \; w_1[d]\, r_2[d]\, c_2\, a_1 \quad s_2 : \; w_1[d]\, a_1\, r_2[d]\, c_2$$

In this paper we make no assumptions about recovery in our account of serializability. In order to do this we must include in the theory of conflicting actions the context of the outcome of the transactions in which these actions appear.
As an example a consider the local schedule

$$r_i[d]\, w_j[d]\, c_i\, a_j$$

We say a type IV conflict exists between t_i and t_j ordering t_i before t_j. To see this consider the different value read by t_i when the accesses are transposed in the schedule

$$w_j[d]\, r_i[d]\, c_i\, a_j$$

[5]If cascading aborts are needed to ensure recoverability a mechanism is assumed to exist to detect when they are required and perform the necessary transaction aborts.

Definition 2 Complete schedules σ and σ' are *conflict equivalent* iff

- σ and σ' have the same actions and
- for each conflict of type $C \in \{I, \ldots, V\}$ involving accesses $r_i[d]\, w_j[d]$ $(w_i[d]\, w_j[d])$ $(w_i[d]\, r_i[d])$ and terminals e_i, e_j, in σ the same conflict of type C appears in σ' involving the same accesses $r_i[d]\, w_j[d]$ $(w_i[d]\, w_j[d])$ $(w_i[d]\, r_i[d])$ and terminals e_i, e_j.

□

Definition 3 A complete schedule σ is *serializable* if it is conflict equivalent to some complete serial schedule.

□

So far we have defined serializability only on complete schedules (i.e. those where all transactions in the schedule eventually either abort or commit). In online transactions processing systems schedules are dynamic objects that never complete. Furthermore failure and recovery can cause schedules to be truncated[6]. For this reason we seek to define serializability over prefixes of complete schedules.

Definition 4 A prefix of a complete schedule is *serializable* if its abort completion is serializable.

□

Serializable schedules isolate transactions from one another. This is because the actions of each transaction, although interleaved, are equivalent (in the sense of conflict resolution) to a serial schedule. This property is very useful as from the transaction's perspective it seems as though it is executing in isolation. Each transaction does not interfere with any other.

Many different implementations of schedulers that allow only serializable schedules have been proposed [4]. Berenson et al. identified a set of phenomena with the property that if none of these phenomena are present in a schedule then that schedule is serializable. These were further extended in [9] and we restate these phenomena as follows:

$$
\begin{array}{llll}
\mathbf{P0}: & w_i[d] \prec w_j[d] \prec e_i & \mathbf{NP1}: & w_i[d] \prec r_j[d] \prec (c_j \wedge a_i) \\
\mathbf{NP2R}: & r_i[d] \prec w_j[d] \prec c_i & \mathbf{NP2L}: & w_i[d] \prec r_j[d] \prec (c_j \wedge c_i)
\end{array}
$$

[6] Nested transactions and synch points allow partial rollback after failure but these constructs are not considered in this paper.

2.3 Strict two phase locking

One way for a scheduler to exclude these phenomena is by implementing strict two-phase locking (S2PC)[7]. Although S2PL guarantees the absence of the phenomena above thus ensuring schedules are serializable it unfortunately prevents schedules in which these phenomena do not occur. In practice however this technique has become widely accepted as a means to ensure serializable isolation.

3 Distributed Schedules

In distributed databases data items are distributed across a set of sites $\mathcal{S}$. We let $SITE(d)$ denote the unique site to which d belongs. Read and write accesses are denoted $r_i[d]$, $w_i[d]$ as before. Once the accesses of a transaction have completed, at a site, they are terminated at that site, we denote these terminals in distributed schedules $c_i[s]$, $a_i[s]$ making reference to the site at which they occur.

The accesses of concurrently executing distributed transactions produce actions that are scheduled locally at each site s. We refer to this schedule as the *local schedule* at s.

Distributed transactions must fulfill the following additional requirements over centralised transactions:

- Distributed transactions must be *atomic*, that is if t_i commits (aborts) at one site it must not abort (commit) at another. This statement is trivially true in the centralised case.
- A distributed transaction must have the ability to unilaterally abort at a site if, for example, an access at that site caused a deadlock or violated an integrity constraint. This means that all the accesses required for a transaction at a site must be completed before that site can determine if it is willing or prepared to commit.

To ensure atomicity amongst transactions a *commit protocol* is used. The most popular of these is the family of two-phase commit protocols. In a 2PC protocol, before any site can commit, that site must acquire (usually by the receipt of messages) knowledge that each site involved in the transaction is willing or prepared to commit. This property is true for all 2PC protocols[7] and was called 2PC-level of knowledge in[8].

[7]Unless some special mechanism, such as compensating transactions is in place.

It follows that from this and the requirement of 2PC-level of knowledge that a transaction t_i *must not* commit causally before[6] any access of t_i at any other site. We call this constraint *causal commitment* and denote it

$$\forall s \in \mathcal{S},\, o_i[d] \rightarrow c_i[s]$$

where $SITE[d]$ need not be s. All distributed schedules must adhere to causal commitment and atomicity.

To see why causal commitment is necessary suppose t_i commits at s then later t_i accesses some item at s' which violates an integrity constraint causing t_i to abort at s'. The transaction has committed at s and aborted at s' violating atomicity.

Definition 5 A *complete distributed schedule*, ρ is the vector of complete local schedules $\rho = (\sigma_{s_1}, \ldots, \sigma_{s_n})$.

□

For notational convenience we write distributed schedules as a list of local schedules. An example is given below, where $s = SITE(d)$ and $s' = SITE(d')$.

$$\begin{array}{llllll} \sigma_s : & r_1[d] & r_2[d] & w_2[d] & c_1[s] & c_2[s] \\ \sigma_{s'} : & r_2[d'] & w_2[d'] & r_1[d'] & c_2[s'] & c_1[s'] \end{array}$$

We use the same notational conventions as we did in centralised schedules, that if one action is to the right of another in the same row we know that the rightmost, of the two actions, took place later. If however two actions are in different rows we can only determine the relative causal order of these actions by using the causal commitment requirement together with the given local order of actions. For example we know that $r_1[d']$ in sequence $\sigma_{s'}$ is before $c_2[s]$ in sequence σ_s by applying transitivity to the facts that $r_1[d']$ must be before $c_1[s]$ by causal commitment and that $c_1[s]$ is before $c_2[s]$ by local order.

Definition 6 A *complete distributed schedule* is *serial* if there exists a total order of the transactions such that if t_i precedes t_j in the order, then all of t_i's actions precede all of t_j's actions in each local schedule where both appear [3].

Definition 7 A prefix of a distributed schedule is the vector of (possibly empty) prefixes of the original local schedules. A prefix $\rho' = (\sigma'_{s_1}, \ldots, \sigma'_{s_n})$ of $\rho = (\sigma_{s_1}, \ldots, \sigma_{s_n})$ is *admissible* provided

$$\text{if } c_i[s_j] \in \sigma'_{s_j} \text{ then } \forall s_k \in \mathcal{S}, \text{ if } o_i[d] \in \sigma_{s_k} \text{ then } o_i[d] \in \sigma'_{s_k}$$

In distributed databases logging to stable storage ensures that the prefix reconstructed on recovery from a failure is always admissible. We will assume therefore that all prefixes are admissible.

Definition 8 The *abort-completion* of a prefix of a distributed schedule ρ, is formed by extending each local schedule σ_s of ρ in the following way. An $a_i[s]$ is appended to σ_s for each transaction t_i with an access but no terminal in σ_s, provided t_i has not committed in any other local schedule of ρ. If it has been committed elsewhere and some access $o_i[d]$ exists in σ_s we append $c_i[s]$ to σ_s.

Clearly, an abort-completed prefix of a complete distributed schedule is a complete distributed schedule. Furthermore, because any prefixes must be admissible, any transaction that commits in the abort-completed prefix must contain all the actions that it did in the original schedule.

4 Serializability of Distributed Schedules

Definition 9 Distributed schedules ρ and ρ' are *conflict equivalent* iff

- ρ and ρ' have the same actions and
- for each conflict of type $C \in \{I, \ldots, V\}$ involving accesses $r_i[d]\, w_j[d]$ $(w_i[d]\, w_j[d])$ $(w_i[d]\, r_j[d])$ and terminals $e_i[s]$, $e_j[s]$, in ρ the same conflict of type C appears in ρ' involving the same accesses $r_i[d]\, w_j[d]$ $(w_i[d]\, w_j[d])$ $(w_i[d]\, r_j[d])$ and terminals $e_i[d]$, $e_j[d]$.

□

Definition 10 A complete distributed schedule ρ is serializable if it is conflict equivalent to some complete distributed serial schedule.

□

Just as in the centralised case we define a prefix of a complete distributed schedule to be serializable iff its abort-completion is serializable.

Proposition 1 All prefixes of complete serializable schedules are serializable.

Proof Sketch Let $\rho = (\sigma_{s_1}, \ldots, \sigma_{s_k})$ be a complete distributed serializable schedule and let ρ_{pref} be any admissible prefix of ρ. Let ρ_{com} be the abort-completion of ρ_{pref}.

If accesses $o_i[d]$ and $o_j[d]$ exist in ρ_{com} then they also exist in ρ and furthermore they are in the same order in each of the schedules. The terminals e_i, e_j in ρ_{com} need not be the same (or be in the same order) as e_i, e_j in ρ. We do however know that if e_i (e_j) is c_i (c_j) in ρ then it will also be c_i (c_j) in ρ_{com}, from the way ρ_{com} is constructed and that transactions are atomic in ρ. It thus follows for each conflict ordering t_i before t_j, in ρ_{com}, at some site s_k, involving accesses $o_i[d]$ and $o_j[d]$, there exists a conflict ordering t_i before t_j involving the same accesses $o_i[d]$, $o_j[d]$ at s_k in ρ. ρ_{com} will therefore have a subset of the conflicts found in ρ. Since ρ is serializable then so is ρ_{com}.

$\square$

A graph theoretic characterisation exists for the conflict serializability of a distributed schedule. Let G be a directed graph whose nodes are the transactions of a schedule ρ. If a conflict exists in a schedule we add a directed edge between $t_i \rightarrow t_j$.

Proposition 2 *G is acyclic* iff *ρ is serializable.*

Proof A similar proof can be found in [4].

Proposition 3 *If a complete distributed schedule ρ is serializable then each complete local schedule is also serializable.*

Proof This follows directly from the fact that if G is acyclic then any sub-graph of G created by restricting the vertex set of G to the subset of transactions that have actions at a particular site will also be acyclic.

$\square$

We can see in the example below (which violates causal commitment) the converse of Proposition 3 does not necessarily hold (*c.f.*[10]).

$$\begin{array}{lllll} \sigma_s : & r_1[d] & c_1[s] & w_2[d] & c_2[s] \\ \sigma_{s'} : & w_2[d'] & c_2[s'] & r_1[d'] & c_1[s'] \end{array}$$

4.1 Local rules for distributed serializability

We have seen that the absence of phenomena **P0**, **NP1**, **NP2L**, **NP2R** give rise to serializable executions of local schedules. However we have seen that local serializability does not imply the serializability of a distributed schedule. We require a further condition to achieve this. It turns out causal commitment is such a condition.

Lemma 1 *Let t_i and t_j be two transactions that conflict on a data item d at site s in a local schedule σ, which we can write generically as*

$$o_i[d] \prec o_j[d] \wedge e_i[s] \wedge e_j[s]$$

If phenomena **P0**, **NP1**, **NP2L**, **NP2R** *do not occur over the actions of this conflict then (1) $e_i[s] \prec o_j[d]$ and (2) $e_i[s] = c_i[s]$*

Proof For a proof see[9].

□

Theorem 1 *Any complete distributed schedule which contains no local phenomena of the type* **P0**, **NP1**, **NP2L** *and* **NP2R** *is serializable.*

Proof If a schedule ρ is not serializable then a cycle of conflicts between transactions must exist. Each conflict must be on a particular data item so we can denote this cycle as $t_1 \xrightarrow{d_1} t_2 \xrightarrow{d_2} \ldots t_m \xrightarrow{d_m} t_1$. Using Lemma 3 and causal commitment, $o_i[d] \rightarrow c_i[s]$, we can derive the following causal order on actions in ρ.

$$\begin{array}{ccccccccccccc}
o_1[d_1] & \prec & c_1[s_1] & \prec & o_2[d_1] & \prec & e_2[d_1] & & & & & & \\
 & & & & \downarrow & & & & & & & & \\
 & & o_2[d_2] & \prec & c_2[s_2] & \prec & o_3[d_2] & \prec & e_3[s_2] & & & & \\
 & & & & & & \downarrow & & & & & & \\
 & & & & o_3[d_3] & \prec & c_3[s_3] & \prec & o_4[d_3] & \prec & e_4[s_3] & & \\
 & & & & & & & \ddots & \ddots & \ddots & & & \\
 & & & & & & & & \downarrow & & & & \\
 & & & & & & o_m[d_m] & \prec & c_m[s_m] & \prec & o_1[d_m] & \prec & c_1[s_m]
\end{array}$$

This gives rise to the contradiction that $o_1[d_m]$ is causally after $c_1[s_1]$ which violates causal commitment.

□

We can conclude S2PL (ensuring the absence of isolation phenomena) together with causal commitment ensures distributed serializability.

5 Lower Isolation Levels

If a transaction releases read locks at a site as soon as a read access has completed at that site, non-serializable schedules are possible for instance the read of inconsistent states in in the well known bank transfer example. This level of isolation is referred to as READ COMMITTED[1] and is supported by many commercial databases[5]. In our phenomena based isolation definition the early release of read locks allows phenomenon **NP2R**.

It is often assumed that if read locks are released early then only this lower isolation level can be achieved. In Proposition 4, below, we show that if transactions that would cause a schedule to be non-serializable, if committed, are instead aborted, then the resulting schedule will be serializable.

Proposition 4 *Suppose all phenomenon except* **NP2R**[8] *are disallowed in a schedule. If transactions abort in all cases where if they were to commit a non-serializable schedule would result, the resulting schedule is guaranteed to remain serializable.*

Proof: Let t_j be a transaction such that if it were to commit it would produce a cycle of in the conflict graph. This must mean that if it were to commit it would order some transaction both before it and after it which we can write as

$$t_i \rightarrow t_j \rightarrow t_k.$$

If aborted the only type of conflict that may exist between t_i and t_j and between t_j and t_k are those of type IV (types I, II and III require both transactions to commit and type V is disallowed by **NP1**). A type IV conflict can order t_i before t_j but cannot order t_j before t_k. We can conclude the cycle cannot exits.

□

6 Modeling Distributed Transactions

In this section we model a very simple distributed database in which concurrent transactions read and write distributed data items. We use

[8] This can be achieved by holding write locks until commit or abort time but releasing read locks after each read operation.

simple locking of data items to implement concurrency control. In our model only write locks are held until a transaction terminates. A read access to a data item is assumed to be atomic but after a read has taken place other transactions may access this data item, even before the reading transaction has terminated. This level of concurrency control would normally result in a level of isolation known as READ COMMITTED admitting non-serializable schedules.

In our scheme, before a transaction commits it checks if it could cause the schedule to be non-serializable, if it could it instead aborts. We show in Theorem 3 that this leads to serializable schedules, thus restoring the higher level of transaction isolation.

The processes in our system are of two types. The set of sites $\mathcal{S}$ and the set of transactions $\mathcal{T}$. We denote a particular transaction or site from these sets s and t respectively.

Each transaction t has the following local state consisting of a set of actions denoted $t.A$. This contains read and write actions for that transaction. For example if t reads $s_1.v$ and writes $s_2.v$ then initially $t.A = \{r[s_1.v], w[s_2.v]\}$. Each transaction carries out at most one action of either type (read or write) at any one site[9]. We use the notation $o[s.v]$ to denote either $r[s.v]$ or $w[s.v]$.

The set of sites where actions of a transaction t take place is denoted $\text{SITES}(t) = \{s \mid w[s.v] \in t.A \text{ or } r[s.v] \in t.A\}$. Actions are assumed to have originated from some external application. In practice transactions may be interactive and have branching control structure. This means that actions are created as the transaction proceeds. We only model the case where all the actions are known at the start of the transaction[10], however the rules we give later, model all possible orders in which they might be executed.

Each site s has the following local state.

1. A data value $s.v$.
2. An exclusive lock $s.X$, for that data value.
3. A set $s.R$ containing all transactions that have read $s.v$ since it was last written to.
4. A set $s.W$ containing the transaction that last wrote $s.v$, initially this contains $\perp$, denoting an imaginary transaction that is assumed to have written all data items.

[9] This restriction is for notational convenience only.

[10] Our protocol does not make use of this knowledge and is therefore applicable in the more general case.

5. An acknowledgement set $s.A$ used to acknowledge transactions' actions.
6. A coloured graph containing any conflicts or potential conflicts[11] of which that site has knowledge. The vertices of the graphs are transactions from the set $\mathcal{T}$. Vertex t is coloured green if it is known that it has committed, red if it is known that it has aborted, and white if it is is not known whether or not it has terminated. Table 6 summarises the state stored at both the site and transaction processes.

Name	Description	Initial Value	Viewable
$s.v$	Data value	0	N
$s.X$	Transaction holding exclusive lock	$\emptyset$	N
$s.W$	Last Transaction to write to $s.v$	$\{\perp\}$	N
$s.R$	Transactions that read since last write	$\emptyset$	N
$s.A$	Acknowledgement set	$\emptyset$	Y
$s.G$	Conflict graph	G_0	N
$t.A$	Actions of t	Initial actions	Y

Table 1: Local state at transaction and site processes. G_0 is the graph with the single green vertex $\perp$ containing no arcs.

6.1 Views and Message Passing

We use the notation $p.v$ to talk about the state variable v at process p. For example, if transaction t holds an exclusive lock at s we say $s.X = \{t\}$.

Processes in our distributed system have their own local state and a *view* of the internal state of other processes. This view is constructed from information they receive from other processes in the form of messages. We say,

$$q.v = \lhd x$$

holds at process p if the most up-to-date message p received from q informed p that q *was* in state x.

[11] A potential conflict exists between transactions that have not both terminated, where possible termination outcomes could lead to a conflict.

Not all local state is exchanged between processes. We distinguish between state variables that can be viewed at remote processes and call these variables *viewable* and those which cannot be seen, and call them *non-viewable*.

We assume that processes in our system are connected by message passing channels. Messages take some time to travel along these channels but are never lost and arrive in the order they were sent[12].

When a process q changes an externally–viewable local state variable, in our notation, $q.v := x$, a message is passed to a remote process p [13] enabling p to update its view of q. After the message is delivered at p, $q.v =\lhd x$ holds at p. We abstract all other communication issues (e.g. message routing, error detection, reliable delivery) in this way.

Sometimes the state stored at q will include more structured data than scalar values. In particular, in our model we wish to store sets of values. We say,

$$q.V \ni\lhd x$$

holds at p where V is a set of values in q's local state. This assertion will hold whenever the most up-to-date message, pertaining to the insertion or deletion of x in V at q, that p has received informs p that q's local state set V contained the element x. When q adds, or removes[14], an element to its local set $q.V$ a message is propagated to other processes allowing them to update their views.

In our protocol the transaction processes collect the coloured graphs $s.G$ from multiple sites and merge them into a single graph. We use the notation $\bigoplus_{s\in\mathcal{S}} s.G$ to denote the graph G_m, constructed by merging each graph $s.G$ at $s \in \mathcal{S}$ in the following way. If an vertex or arc exists in one of the graphs $s.G$ that is not present in G_m it is added to G_m. If a vertex is green (red) in *any* of the graphs $s.G$ then it will be green (red) in G_m. In our protocol a vertex can never be green in one graph and red in another.

The merged graph $\bigoplus_{s\in\mathcal{S}} s.G$ is composed from the local graphs $s.G$ at each site. These graphs may be updated as messages propagate between sites. For this reason no site can guarantee that it knows this information. We can however construct the *merged view*. We say,

$$s.G \bigoplus_{s\in\mathcal{S}}\lhd G$$

[12] This level of message passing service is achievable in practice.

[13] We don't specify to which processes messages are sent preferring to abstract this to those processes that are interested.

[14] Sometimes messages are sent only when elements are added to a set.

holds at t if views of $s.G$ from all sites $s \in S$ have been merged to give the value G.

6.2 Rules, State and Executions

Rules determine the behaviour of a process by transforming the state of that process given certain local conditions are met. A rule is a pair which consists of a pre-condition and a postaction. The pre-condition is a logical statement which may contain assertions about views of remote process states. A postaction is a list of assignments to local state variables; assignments to viewable variables cause messages to be sent to remote processes. The clauses of the postactions are carried out in left to right order.

The global state of our system is formed by composing the local state of each transaction and each site process $\omega = (\mathcal{S}, \mathcal{T})$. If a transaction or site process takes a step and changes local state then the global state takes a step. An *execution* of our protocol is a sequence of global state changes $\omega_0, \omega_1, \ldots$.

6.3 The Rules of Our Protocol

Rules determine the behaviour of each process and thus the possible executions of the protocol. One set of rules is given for transaction type processes and another for site type processes. Each rule is a template for either a transaction t or a site s. For example, our first rule describes the actions taken at a site s when its view of some transaction t is updated, causing s to carry out a read or write action.

$$\frac{t.A \ni \lhd o[s.v] \;\wedge\; (s.X = \emptyset \;\vee\; s.X = \{t\}) \;\wedge\; ack_t \notin s.A}{\text{UPDATE}(s.G, t, o[s.v]) \;\wedge\; s.A \cup = \{ack_t\}}$$

$$\text{UPDATE}(G, t, o[s.v]) = \begin{cases} s.G \cup = \{t' \to t \,|\, t' \in s.W \cup s.R \;\wedge\; t' \neq t\} & \\ \wedge\, SAVE(s) \;\wedge\; s.X := \{t\} \wedge & \text{if } o = w \\ s.R := \emptyset \;\wedge\; s.W = \{t\} & \\ & \\ s.G \cup = \{t' \to t \,|\, t' \in s.W \;\wedge\; t \neq t'\} \wedge & \text{if } o = r \\ s.R \cup = \{t\} & \end{cases}$$

$$\text{SAVE}(s) \stackrel{def}{=} s.R_s := s.R \;\wedge\; s.W_s := s.W \;\wedge\; s.v_s := s.v$$

[15] The next rule describes the behaviour of a transaction process, t, when it receives an acknowledgement that an action has taken place.

$$\frac{o[s.v] \in t.A \;\wedge\; s.A \ni\!\lhd ack_t}{t.A := t.A - \{o[s.v]\}}$$

For notational convenience we define the following sets. $\mathrm{PREPARE} \stackrel{def}{=} \{p_s \mid s \in \mathcal{S}\}$, $\mathrm{COMMIT_t}(G) \stackrel{def}{=} \{c_s^G \mid s \in \mathit{SITES}(t)\}$ and $\mathrm{ABORT_t}(G) \stackrel{def}{=} \{a_s^G \mid s \in \mathit{SITES}(t)\}$. The following rule describes the action the transaction process takes when it has acknowledgements that all its actions have completed.

$$\frac{t.A = \emptyset}{t.A := \mathrm{PREPARE}}$$

When a site sees a request to prepare p_s it returns a snapshot of its current conflict graph $s.G$ this behaviour is captured by the rule

$$\frac{t.A \ni\!\lhd p_s}{s.G_t := s.G}\,^{16}$$

The next two rules are perhaps the most important in our protocol. They describe how a transaction process determines if the transaction should be committed or aborted based on the conflict graphs it collects from each site. As the graphs are collected they are merged to create a conglomerate view. Two graphs are merged in the usual way but if a vertex t is green (red) in one graph and white in the other then in the merged graph it will be green (red) not white. We call these two rules the Commit and Abort Rules.

$$\frac{s.G_t \bigoplus_{s\in\mathcal{S}}\!\lhd G \;\wedge\; \mathrm{VALID}(G,t)}{\mathrm{COLOUR(G,t,green)} \;\wedge\; t.A := \mathrm{COMMIT_t}(G)}$$

$$\frac{s.G_t \bigoplus_{s\in\mathcal{S}}\!\lhd G \;\wedge\; \neg\mathrm{VALID}(G,t)}{\mathrm{COLOUR(G,t,red)} \;\wedge\; t.A := \mathrm{ABORT_t}(G)}$$

Where $\mathrm{VALID}(G,t) \stackrel{def}{=} \forall c \in \mathit{cycles}(G,t)\, t \neq \mathit{min}(\{t' \in c \mid \mathrm{COLOUR}(t') = \mathrm{white}\})$, where $\mathit{cycles}(t,G)$ is the set of all cycle through t in G or the emptyset if no such cycle exists. Finally the last two rules allow a site

[15] $s.R_s$, $s.W_s$, $s.v_s$ are non-viewable state variable at each site that allow aborting transactions to undo changes made when a transaction aborts.

[16] Strictly speaking $s.G_t$ should be included in the state of s as an viewable variable. However we can think of this action as $s.G$ being made momentarily viewable to t.

to take appropriate action when it knows the outcome of the transaction.

$$\frac{t.A \ni \lhd c_s^G}{s.G := G \land s.X := \emptyset} \quad \frac{t.A \ni \lhd a_s^G}{s.G := G \land s.X := \emptyset \land \mathrm{UNDO}(s)}$$

$$\mathrm{UNDO}(s) \stackrel{def}{=} s.R := s.R_s \land s.W := s.W_s \land s.v := s.v_s$$

6.4 Protocol Correctness

Definition 11 We define M to be the set of vertices *missing* in a graph $G = (V, A)$ with respect to a cycle c *as*

$$M \stackrel{def}{=} \{t \mid \exists t',\, (t' \to t) \in c \land (t' \to t) \notin A\}$$

Theorem 2 *In any execution of our protocol at least one vertex in any cycle formed in* $G_m = \bigoplus_{s \in \mathcal{S}} s.G$ *is white.*

Proof: Suppose a cycle forms in G_m after some step in an execution, furthermore assume for the sake of contradiction that each vertex in this cycle is green.

For this to happen the same cycle must have existed in $\bigoplus_{s \in \mathcal{S}} s.G$ immediately after an earlier step but in this cycle some of the vertices are white. [To see this note that the first time a cycle is completed it is after a read or write rule that adds an arc to a local $s.G$. At least one end point of this arc must be white as the transaction performing the read or write has not yet committed.].

Let this cycle be $c = t_0 \to \ldots \to t_{l-1} \to t_l = t_0$ for $l \geq 1$ WLOG suppose t_0 is the smallest vertex in the cycle amongst the white vertices.

Now consider the point in the execution where t_0 is first coloured green. (This must be in the postaction of Commit rule in the $\mathrm{COLOUR}(G, t_0, green)$ clause.). The commit rule for t_0 is

$$\frac{s.G_{t_0} \bigoplus_{s \in \mathcal{S}} \lhd G^0 \land \mathrm{VALID}(G^0, t_0)}{\mathrm{COLOUR(G, t_0, green)} \land t_0.A := \mathrm{COMMIT}_{t_0}(G)}$$

When the pre-condition is evaluated the cycle c cannot be in the merged view G^0. If it was then $VALID(G^0, t_0)$ will not hold. Let M^0 be the set of vertices missing in the merged view, G^0, with respect to c. And let t^1 be the smallest white vertex in M^0. Such a vertex must exist as if all vertices in M^0 are green then c is in G^0. (To see this note when a vertex is coloured green the local graph will contain all its parents and these graphs will be in the constructed view G^0).

Now consider the later point in the execution at which t^1 is first coloured green. We construct G^1 and M^1 in the same way as G^0 and M^0. Now t_0 must be green in G^1. This is because t^1 had not sent a prepare message to $t_0.s$ before sending its part of G^0 to t, and so its prepare message must arrive at $t.s$ after t was coloured green. Furthermore, no new vertices are in M^1 that were not in M^0 because G_1 contains every arc that G_0 did by the same argument. This means the number of white vertices in M^1 is smaller than those in M^0. We can now take the smallest of these white vertices t^2 and re-apply the argument removing it from M^1.

After re-application of this process enough times no vertex in the missing set can be white. and at this point the cycle c is in the constructed view. This means $VALID(G,t)$ will eventually fail when committing some transaction on the cycle and this transaction cannot therefore be coloured green.

□

Theorem 3 *All executions of our protocol produce serializable schedules.*

Proof: From Theorem 2 no conflict cycles exist between committed transactions. Therefore a topological sort of the conflict graph produces a serial order for the execution of each transaction that is conflict equivalent to the actual schedule produced.

□

7 Conclusions

We have shown how isolation levels in centralised schedules can be defined by the absence of different phenomena. We generalised this notion to distributed schedules. Locking is often used to preclude phenomena from schedules. Many database vendors provide a lower level of isolation by allowing the early release of read locks. By sending additional information during the commit protocol of a distributed transaction lower isolation levels can be detected and raised to full isolation.

References

[1] ANSI x3.135-1992. *American National Standard for Information Systems–Database Language–SQL*, November 1992.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(4), 1995.

[3] P.A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

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

[5] Microsoft Corporation. *Microsoft Distributed Transaction Coordinator Resource Manager Implementation Guide*, January 1996. Version 6.5.

[6] L.J. Fidge. Logical time in distributed computing systems. In *IEEE Computer*, pages 28–33. IEEE Comput. Soc. Press, August 1991.

[7] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, CA, 1993.

[8] V. Hadzilacos. A knowledge theoretic analysis of atomic commitment protocols. In *ACM Principles of Database Systems (PODS)*, pages 129–134, 1987.

[9] T. Kempster, C. Stirling, and P. Thanisch. Diluting ACID. Technical Report ICSA-47-99, Dept. of Computer Science, University of Edinburgh, Febuary 1999. On-line, http://www.dcs.ed.ac.uk/home/tdk/.

[10] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1991.