Diluting ACID

Tim Kempster*, Colin Stirling, Peter Thanisch
Division of Informatics, University of Edinburgh

Abstract

Several DBMS vendors have implemented the ANSI standard SQL isolation levels for transaction processing. This has created a gap between database practice and textbook accounts of transaction processing which simply equate isolation with serializability.

We extend the notion of conflict to cover lower isolation levels and we present improved characterisations of classes of schedules achieving these levels.

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.

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[1]. 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_2, read a value for a data item that was written by another transaction, t_1, *before* t_1 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.

When defining isolation levels, the ANSI standard [6] [1] deliberately avoids reference to locking, thereby making the standard relevant to non-locking based concurrency control algorithms. It instead defines isolation levels by specifying types of "phenomena" which are disallowed if a particular isolation level is to be achieved.

Berenson et al. [2] criticise the ANSI standard. They highlight some serious shortcomings and provide alternative isolation level definitions based on locking. Berenson et al. also provide a definition, based on phenomena, that they claim is equivalent to their locking based definition. In this paper we demonstrate that their phenomena based definition is not equivalent to their locking based definition.

Many textbooks state that isolation and serializability are synonymous [4] [5]. However we argue in this paper that isolation is really a sufficient but not necessary condition for serializability. Indeed, the isolation levels defined in the literature exclude many serializable schedules.

The main contributions of this paper are threefold. Firstly, we provide a definition of conflicting actions that includes the commit or abort outcome of the transactions involved in the conflict. Under this extended definition we need not make any recoverability

*Rm 2602, JCMB, Kings Buildings, University of Edinburgh, Mayfield Road, Edinburgh, EH9 3JZ, Scotland Tel +44 131 650 5139, Fax +44 131 667 7209. This research was supported by EPSRC grant GR/L74798.

[1] 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.

assumptions about schedules. This leads to a definition of conflict serializability that is independent of recoverability. This definition of conflict serializability does not include the phantom phenomenon (discussed later) because it is defined over schedules that do not include reads and writes over predicates. It is therefore equivalent to the ANSI definition REPEATABLE READ.

Secondly, we provide a definition of the lowest three ANSI isolation levels based on phenomena. Our phenomena are weaker than those proposed in [2] and thus admit more serializable schedules.

Lastly, we enrich schedules to include read and write actions on predicates. Within these enriched schedules we discuss Phantom Phenomena and characterise them in a way that is independent of predicate concurrency control mechanisms. Excluding these phantom phenomena results in a level of isolation termed SERIALIZABLE by the ANSI community.

2 Schedules

Let t_i, t_j denote transactions and d, d' denote data items in the 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, consists of actions. These actions are divided into four categories. Read and write actions which we call *accesses* and denote r_i, w_i respectively or o_i to denote either a r_i or w_i, together with 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. When a transaction commits, the changes it has made to the data items of the database are made durable, and the values it has read are returned to the user. If a transaction aborts, all write actions are undone leaving any data items with the value that they would have had if the transaction had never executed, furthermore no read values are returned.

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] and also that exactly one kind of terminal for each transaction occurs exactly once[3]. A *schedule*, s, is a sequence[4] of actions generated by a scheduler. We say $o_i[d] \prec o'_j[d]$ if an action $o_i[d]$ is earlier than an action $o'_j[d]$ in s. In any schedule no terminal of a transaction precedes an action of that transaction. An example of a schedule is $w_1[d]\, r_2[d]\, w_1[d']\, c_1\, c_2$. A *serial schedule* is one in which all actions of one transaction are completed before any action of another transaction is started.

By slightly abusing notation we say c_i (a_i) is true over a schedule if action c_i (a_i) happens at some point. 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 but does so after a write action by t_i. 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 $w_j[d]$. It should be noted this allows a_i before or after $w_j[d]$.

3 Conflict Serializability

To define serializability we must first define equivalence over schedules. The most common and useful definition is that of conflict equivalence [3]. Unfortunately, this definition fails to capture the inequivalence of schedules containing aborting transactions. For example, in this definition the following two schedules are defined to be equivalent.

$$w_1[x]\, r_2[x]\, a_1\, c_2 \;\equiv\; w_1[x]\, a_1\, r_2[x]\, c_2$$

The classical definition of conflict equivalence requires the ordering of conflicting actions from committing transactions to be maintained, but says nothing about the ordering of actions of aborting transac-

[2]The results in this paper do not depend on this but it is a useful notational convenience.

[3]Schedules with this property are often called *complete schedules*.

[4]A schedule is also sometimes defined as a poset of actions, and sometimes called a history. We choose to define it as a sequence in order to keep it consistent with [2].

tions. To capture this behavior we extend the classical definition of conflict equivalence by first extending the definition of a conflict.

Two transactions t_i and t_j are said to *conflict* on a data item, d, if both access d and transposing the order of these 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. We enumerate all the possible types of conflict that can occur in a schedule below.

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

We can see that we have extended the classical notion of a conflict to include the context of the outcome of the transactions. If we remove this context the five conflict types collapse into the classical three conflict types: read/write, write/write and write/read.

At first sight type IV might not look like a conflict. If we transpose the accesses we arrive at either

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

or

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

The first transposition does not change the values read or written by the transactions but in the second case it does. However if recoverability mechanisms are assumed to be in place the second type of schedule could not occur because in the case $a_j \prec c_i$ t_j's abort action would cascade and cause t_i to abort and in the case $c_i \prec a_j$, t_i cannot be allowed to commit until t_j had terminated. Type IV conflicts are included because we seek a definition that does not make assumptions about recovery.
The schedule

$$r_1[d]\, w_2[d]\, w_2[d']\, r_1[d']\, c_1\, a_2$$

has two conflicts, the first between $r_1[d]$ and $w_2[d]$, an instance of IV above, and the second between $w_2[d']$ and $r_1[d']$ which is an instance of V above. The extended definition of conflict equivalence naturally follows from the extended definitions of conflicts.

Definition 1 Schedules s and s' are *conflict equivalent* iff

- s and s' have the same actions and
- for each conflict of type $C \in \{I, \ldots, V\}$ involving actions o_i, o_j, e_i, e_j, in s the same conflict of type C appears in s' involving the same actions o_i, o_j, e_i, e_j.

□

Definition 2 Schedule s is conflict serializable if it is conflict equivalent to some serial schedule.

□

Our definition of conflict serializability coincides with Bernstein et al. [3], in the case when the committed projection of schedules is considered. However we can now judge equality between schedules containing aborting transactions. For example, under our new definition

$$w_1[x]\, r_2[x]\, a_1\, c_2 \not\equiv w_1[x]\, a_1\, r_2[x]\, c_2$$

because the write-read conflict (type V) on the left hand side does not exist on the right hand side.

Although conflict serializability has been defined only on complete schedules (i.e. those where all transactions in the schedule eventually either abort or commit) we can extend the definition to incomplete schedules. Any incomplete schedule can be extended to a complete schedule by aborting all the transactions without a terminal, we call the resulting schedule the *aborting-completion*. We now say an incomplete schedule is serializable iff its aborting-completion is serializable.

In real systems failures can truncate schedules at any point. Upon recovery active transactions are aborted. For this reason a useful property of any serializability definition over schedules is that if p is a prefix of a serializable schedule s, then p is serializable.

Proposition 1 Any prefix of a complete conflict serializable schedule is conflict serializable.

Proof Let s denote a complete serializable schedule and s_{ser} denote a serial schedule that is conflict equivalent to s. Let p denote any prefix of s. We will construct a serial schedule p_{ser} from s_{ser} that is conflict equivalent to the aborting-completion of p, which we denote p_{com}. This shows that any prefix of a complete serializable schedule is serializable. We do this in two steps.

S1 For each $a_i \in p_{com}$ such that $c_i \in s_{ser}$, replace the c_i in s_{ser} with a_i to form p'.

S2 If any action appears in p' but not in p_{com} remove the action from p' to form p_{ser}.

We will now show that p_{ser} is a serial schedule that is conflict equivalent to p_{com}. Clearly p_{ser} is serial and has the same actions as p_{com} because of the way it was constructed from s_{ser}. We must now show that if a conflict of type C appears in p_{com} it also appears in p_{ser}.

If p_{com} has a conflict of type I, II or III, then it will also be in s_{ser} because it was in s. Steps **S1** and **S2** will not remove this conflict so it will also be present in p_{ser}.

Suppose p_{com} has a conflict of type IV or V, then either it was in p, and by a similar argument to the one above will be in p_{ser}, or a new conflict will have been formed when the abort completion of p was taken to give p_{com}. If a new conflict was formed of type IV (V) in p_{com} then a conflict of type I (II) must have been present in s so it will also be present in s_{ser} and will be changed to a conflict of type IV (V) by step **S1** when constructing p_{ser} as required.

□

4 Redefining Phenomena

As pointed out by Berenson et al. [2] the phenomena based definitions of isolation levels proposed in the ANSI standard [1] are ambiguous and incomplete. They give more precise definitions in response to these deficiencies. We restate these improvements in our notation and extend them a little further.

Berenson et al. [2] considered two possible interpretations of the ANSI Dirty Read phenomenon; a strict (**P1** below) and a loose interpretation. They argued that the strict interpretation was required to prevent the classical inconsistent analysis problem exemplified in the history H1 below.

$$H1: \quad r_1[x=50]w_1[x=10]r_2[x=10]r_2[y=50] \\ c_2 r_1[y=50]w_1[y=90]c_1$$

$$\mathbf{P1}: \quad w_i[d] \prec r_j[d] \prec e_i$$

Clearly, the intention is to disallow the situation where t_j reads the changes made by t_i before they are committed. However, it is not always unsafe to do so. In fact, it is only unsafe in the case that t_i aborts after t_j read d and also when t_j commits. For example, consider the serializable schedule $w_i[d]\, r_j[d]\, c_i\, a_j$ which is disallowed by **P1**.

We propose that the loose interpretation of the phenomenon, below, more accurately captures the idea of a Dirty Read. We rename this **NP1** for consistency but it is identical to the loose interpretation called **A1** in [2].

$$\mathbf{NP1}: \quad w_i[d] \prec r_j[d] \prec (c_j \wedge a_i)$$

Unfortunately, the loose interpretation still admits schedules with the inconsistent analysis problem exemplified by history H1. This problem is better captured by the introduction of a new phenomenon we call **NP2L** (see below). Furthermore, we argue that this phenomenon should be disallowed at the higher ANSI REPEATABLE READ isolation level but not at the READ COMMITTED level c.f. [2]. The inconsistent analysis problem arises from transaction t_2 reading an inconsistent view of the database items x and y. Item x is read after t_1 has updated it and y is read before t_1 has updated it. The problem therefore is better described as a Fuzzy or Non-Repeatable read not as a Dirty Read. From a user's perspective the value read by t_2 in H1 is not one that is later aborted as in the case of a Dirty Read. Rather the values reflect partial changes made by other transactions. It should therefore be admitted at the READ COMMITTED level but excluded at the REPEATABLE READ level. Another example of the Fuzzy

read problem appears in H2, a history that is symmetric to H1.

$H2:\quad r_2[x=50] r_1[x=50] w_1[x=10] r_1[y=50]$
$w_1[y=90] c_1 r_2[y=90] c_2$

To prevent this problem Berenson et al. defined phenomena **P2** which we state below.

P2 : $r_i[d] \prec w_j[d] \prec e_i$

Again the intention is to prevent inconsistent reads of database items by ensuring no other transaction t_j may change the value of a data item once read by t_i until after t_i has terminated. It is not always unsafe to do this. For example the schedule, $r_i[d]\, w_j[d]\, a_i\, c_j$, is serializable but not allowed by **P2**.

In our definition we replace **P2** with two phenomena **NP2R** and **NP2L** to capture the two symmetric phenomena that lead to Fuzzy reads of database items. **NP2L** captures the problems of inconsistent analysis found in H1 (thus allowing us to use the loose interpretation **NP1** admitting more schedules at the lower ANSI READ COMMITTED level) and **NP2R** captures the Fuzzy read problem of H2.

NP2R : $r_i[d] \prec w_j[d] \prec (c_i \wedge c_j)$

NP2L : $w_i[d] \prec r_j[d] \prec (c_i \wedge c_j)$

Although excluding phenomena **NP2L**, and **NP2R** from schedules allows more serializable schedules than disallowing **P2**, they still disallow some serializable schedules. For example, the schedule $r_i[d]\, w_j[d]\, c_i\, c_j$ is serializable but disallowed by **NP2R**. This raises the following question. Can we simply characterise using our notation a phenomena that captures only the schedules that read inconsistent views and no more? The answer to this is no. Such a definition would need to include reachability in the associated conflict graph of a schedule. This type of property is not expressible in our notation.

The ANSI standard did not disallow schedules containing so called "dirty writes". This was identified and correctly rectified by the addition of the **P0** Phenomenon in [2]. This phenomenon can also be weakened to **NP0** (below) if we are only interested in isolation properties. In practice its stricter form **P0** is more useful for recoverability and consistency reasons.

P0 : $w_i[d] \prec w_j[d] \prec e_i$

NP0 : $w_i[d] \prec w_j[d] \prec (c_i \wedge c_j)$

Using these phenomena we provide definitions for the lowest three isolations levels; see Table 1.

5 Disallowing Phenomena Provides Conflict Serializability

We now show that if a schedule exhibits none of the phenomena **NP0**, **NP1**, **NP2L** or **NP2R** then it will be conflict serializable as defined in Definition[5] 2. We first prove the following lemma.

Lemma 1 *If a conflict exists between two transactions, t_i and t_j ($t_i \neq t_j$), on data item d which we can write generically as*

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

in a schedule s and phenomena **NP0**, **NP1**, **NP2L**, **NP2R** *do not occur over the actions of this conflict then either ($e_i \prec o_j[d]$ and $e_i = c_i$) or $e_j = a_j$.*

Proof By case analysis of the types of conflict.

I $r_i[d] \prec w_j[d] \wedge (c_i \wedge c_j)$ but **NP2R** does not occur so $c_i \prec w_j[d]$, as required.

II $w_i[d] \prec r_j[d] \wedge (c_i \wedge c_j)$ but **NP2L** does not occur so $c_i \prec r_j[d]$, as required.

III $w_i[d] \prec w_j[d] \wedge (c_i \wedge c_j)$ but **NP0** does not occur so $c_i \prec w_j[d]$, as required.

IV $r_i[d] \prec w_j[d] \wedge (c_i \wedge a_j)$ but $e_j = a_j$, as required.

V $w_i[d] \prec r_j[d] \prec (a_i \wedge c_j)$ but **NP1** does not occur which rules out this type of conflict completely.

Lemma 2 *If transaction t_i aborts in a schedule s that contains no* **NP1** *phenomena then no conflict can exist in s that orders a different transaction, say t_j, after t_i.*

Proof The only possible conflict candidate to order t_j after t_i is a conflict of type V but this conflict is excluded by **NP1**.

[5] This is equivalent to the ANSI Isolation level REPEATABLE READ.

Theorem 1 *All schedules, s, which do not exhibit phenomena* **NP0**, **NP1**, **NP2L**, **NP2R** *are conflict serializable.*

Proof Suppose s is not conflict serializable. Let $G = (V, E)$ be the conflict graph constructed from s as follows. The vertices of G are the transactions in s and an edge (t_i, t_j) is in E if there is a conflict between t_i and t_j ($t_i \neq t_j$) and the accesses of this conflict are ordered $o_i[d] \prec o_j[d]$. Clearly, s is serializable iff G is acyclic (A proof would be similar to Theorem 2.1 [3]).

Suppose s is not serializable without loss of generality let the smallest cycle in the conflict graph G be denoted by

$$t_1 \xrightarrow{d_1} t_2 \xrightarrow{d_2} \ldots \xrightarrow{d_{m-1}} t_m \xrightarrow{d_m} t_1$$

By Lemma 2 no conflict ordering $t_i \xrightarrow{d_i} t_{i+1} : 1 \leq i < m$ in the cycle exists where $e_{i+1} = a_{i+1}$ (so all conflicting transactions in the cycle commit).

By Lemma 1 in each conflict $c_i \prec o_{i+1}[d] : 1 \leq 1 < m$ and also $o_i \prec c_i : 1 \leq i \leq m$ because all actions must be before their terminals, thus we can order the conflicts in the graph as follows.

$$o_1[d_1] \prec e_1 \prec o_2[d_1] \prec e_2 \prec o_2[d_2] \prec e_3 \ldots \prec e_m \prec o_1[d_m]$$

This leads to a contradiction since $e_1 \prec o_1[d_m]$ may not occur in s, so s is serializable.

□

6 Enriching Schedules with Predicate Accesses

We now extend our model with some new types of accesses. Given a predicate P we add a new action, $r_i[P]$, to denote a read of the set of data items that fulfill P. For example, P might be "all employees that are male", so that $r_i[P]$ denotes transaction t_i reading all those employees that are male. We also add two types of write actions $w_i[insert\ y\ in\ P]$ and $w_i[delete\ y\ in\ P]$, these denote actions that insert or delete a new data item, y, in a way that could change the values returned by a $r_i[P]$ access[6]. We write $w_i[y\ in\ P]$ to denote either an insert or a delete access. In our example above $w_i[y\ in\ P]$ might be inserting or deleting a male employee. In this extended model a phenomenon known as phantoms may occur. We restate an example from [2] that exemplifies this phenomenon.

Example 1 *Transaction t_i performs a* `<search-condition>` *to find the list of active employees. Then transaction t_j performs an insert of a new active employee and then updates d', the count of employees in the company. Following this t_i reads the count of employees as a check and finds a discrepancy. The schedule can be written as:-*

$$r_i[P]\, w_j[insert\ d\ in\ P]\, r_j[d']\, w_j[d']\, c_j\, r_i[d']\, c_i$$

In order to characterise this phantom phenomenon Berenson et al. [2] provide the following definition which we restate in our notation as follows.

P3 : $r_i[P] \prec w_j[d\ in\ P] \prec e_i$

Strictly speaking this does not completely characterise all phantom phenomena. Consider Example 2 below.

Example 2 *Transaction t_i deletes an active employee. Transaction t_j then reads the count of active employees z, this will include the one previously deleted by t_i. Transaction t_j then reads the set of all active employees, this will not include the employee deleted by t_i, and then commits. Finally t_i updates the count of new employees and commits. The schedules can be written as follows.*

$$w_i[delete\ y\ in\ P]\, r_j[z]\, r_j[P]\, c_j\, r_i[z]\, w_i[z]\, c_i$$

The schedule of Example 2 contains a phantom read but it is allowed by **P3** therefore strictly speaking the characterisation of phantom phenomena **P3** given by Berenson et al. appears to permit some

[6]Item y does not have to directly satisfy P for this to be true.

kinds of phantoms. Furthermore, it is claimed this phenomena based definition is equivalent the locking based definition of serializable isolation LOCKING SERIALIZABLE they give [2]. In this definition predicate write locks are not released until the transaction commits or aborts, which would prevent the problem in Example 2.

It appears that Berenson et al.'s [2] phenomenon based definition of SERIALIZABLE isolation admits schedules with phantoms, and that it is not equivalent to the locking based definition they provide which does not allow phantoms.

This discrepancy seems to originate from an assumption that a predicate read operation $r[P]$ will conflict with any previous writes (be they deletes or inserts) to data items satisfying the predicate. Implementations do exist to detect exactly this. In some a flag is set in all index entries when a row is deleted, this flag is later garbage collected. Similar implementation details could solve the problem exemplified in Example 1. Rather than make reference to implementations it seems more sensible to define a complete set of phenomena that capture the behavior of both examples. We therefore define phenomena **NP3R**, and **NP3L** in an analogous way to **NP2R** and **NP2L** as alternatives to **P3**. We also define a predicate form of the dirty read and dirty write phenomena which we call **NP2$\frac{1}{2}$** and **NP2$\frac{1}{4}$** respectively.

NP3R : $r_i[P] \prec w_j[d \; in \; P] \prec (c_i \wedge c_j)$

NP3L : $w_i[d \; in \; P] \prec r_j[P] \prec (c_i \wedge c_j)$

NP2$\frac{1}{2}$: $w_i[d \; in \; P] \prec r_j[P] \prec (c_j \wedge a_i)$

NP2$\frac{1}{4}$: $w_i[d \; in \; P] \prec w_j[d \; in \; P] \prec (c_i \wedge c_j)$

We are now in a position to define isolation levels in terms of all our new phenomena see Table 1.

Isolation Level	P0 NP2$\frac{1}{4}$	NP1	NP2R NP2L	NP3R NP3L NP2$\frac{1}{2}$
READ UNCOMMITTED	–	+	+	+
READ COMMITTED	–	–	+	+
REPEATABLE READ	–	–	–	+
SERIALIZABLE	–	–	–	–

Table 1: Definition of isolation levels. [+] denotes a phenomena that is allowable at a particular isolation level whereas [–] denotes that the phenomena is not allowed in any schedules achieving this isolation level.

7 Conclusion

We provided a phenomenon based definition for isolation levels which is also applicable to non-locking based schedulers. Our definition admits serializable schedules which are excluded by the definitions given in [2]. We have shown that if all our isolation requirements are met, schedules will be serializable under our extended definition.

The authors would like to thank the editor and referees for their useful comments and criticisms during the preparation of this paper.

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, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[4] P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing.* Morgan-Kaufmann, San Mateo, CA, 1997.

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

[6] J. Melton and R. Simon. *Understanding the New SQL: A Complete Guide.* Morgan-Kaufmann, San Mateo, CA, 1993.