Diluting ACID

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

Technical Report ECS-LFCS-99-404
Department of Computer Science
University of Edinburgh

March 15, 1999

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 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.

*Rm 2602, JCMB, Kings Buildings, University of Edinburgh, Mayfield Road, Edinburgh, EH9 3JZ, Scotland Tel +44 131 650 5139, Fax +44 131 667 7209.

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.

Arguably, the lower ANSI isolation levels have an indirect impact on consistency (the 'C' in ACID). Suppose that a transaction, say t_i, reads data values that were created by a transaction that subsequently aborts, or it read two data values that belong to two different consistent states. If, subsequently, t_i writes a data item, then the value written is a function of non-existent data. The system could only be said to obey the consistency property if the transactions such as t_i have been designed in such a way that they do not corrupt the database even if they see no consistent state, or a multiplicity of consistent states, in the database.

The classical theory of serializability does 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.

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

To make this distinction the classical theory must be extended. We achieve this by extending the theory of conflicting actions to contain 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 provides alternative isolation level definitions based on locking. Berenson et al. also provide an equivalent phenomenon-based definition to their locking definition. In this paper we demonstrate that their phenomenon 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 by Berenson et al. exclude many serializable schedules.

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, w_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[1] and also that exactly one kind of terminal for each transaction occurs exactly once[2]. A *schedule*, s, is a sequence[3] 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 Extending 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 transactions. To capture this behavior we extend the classical definition of conflict equivalence by first extending the definition of a conflict.

In the classical theory of conflicting accesses, two accesses are said to *conflict* if they both access the same data item and at least one of them is a write. This is expressed in our notation as follows. We relax our assumption that $t_i \neq t_j$ in the definition of conflicts.

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

We can extend the notion of a conflict to cover the case in which, subsequently, one of the transactions aborts. There are two new instances of conflict which capture conflicting actions in schedules containing aborting transactions.

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

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

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 conflict.

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

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

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

[3] 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].

- 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 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 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 serializable schedule is 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}.

Suppose p_{com} has a conflict of type I, II or III, then 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 in by Berenson et al. [2] the phenomena based definitions of isolation levels proposed in the ANSI standard [1] are ambiguous and incomplete. They give much more precise definitions in response to these deficiencies. We restate these improvements in our notation and extend them further.

Berenson et al.'s [2] definition of the dirty read phenomenon may be restated in our notation as follows

P1 : $w_i[d] \prec r_j[d] \prec e_i$

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 the following definition of the phenomenon to capture more accurately the idea of a dirty read.

NP1 : $w_i[d] \prec r_j[d] \prec (c_j \wedge a_i)$

The name non-repeatable read suggests a phenomenon where a read is repeated in a transaction yielding different values brought about by the interference of another transaction. Although this is an example of what might happen we prefer to think of this phenomenon as an inconsistent view. Berenson et al.'s [2] definition of this phenomenon can be restated in our notation as follows.

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

Again the intention is to prevent non-repeatable reads 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**. The non-repeatable read phenomenon can occur in two ways depending on the order of the read and write. We thus propose the two phenomenon to more accurately capture the notion of a non-repeatable read.

NP2R : $r_i[d] \prec w_j[d] \prec c_i$

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

Although excluding phenomena **NP2L**, and **NP2R** from schedules allows more 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. This is because such 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** Phenomena by [2]. We restate and adopt this as a necessary phenomenon in our definitions.

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

5 Serializability

We now show that any if schedules do not exhibit any of the phenomena **P0**, **NP1**, **NP2L** or **NP2R** they will be serializable. 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 **P0**, **NP1**, **NP2L**, **NP2R** *do not occur over the actions of this conflict then $e_i \prec o_j[d]$.*

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 **P0** 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 **NP2R** does not occur so $c_i \prec w_j[d]$, 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.

Theorem 1 *All schedules, s, which do not exhibit phenomena* **P0**, **NP1**, **NP2L**, **NP2R** *are serializable.*

Proof Suppose s is not 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 $o_i[d] \prec o_j$. 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

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

Using both the fact that all accesses of transactions are before their terminals, $o_i[d] \prec e_i$, and also Lemma 1 alternately, we can order the actions of the conflicts which form the cycle in the conflict 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 Predicates

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 $w_i[\mathit{insert}\ y\ \mathit{in}\ P]$ and $w_i[\mathit{delete}\ y\ \mathit{in}\ P]$, these denote actions that insert or delete a new data item item, y, in a way that would change the values returned by a $r_i[P]$ access [4]. We write $w_i[y\ \mathit{in}\ P]$ to denote either an insert or a delete access. In our example above $w_i[y\ \mathit{in}\ P]$ 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:-*

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

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

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

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

Unfortunately, this does not completely characterise phantom phenomena. Consider Example 2.

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

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

The schedule of Example 2 is not serializable but it is allowed by **P3** therefore the phenomenon based definition given by Berenson et al. [2] of a level of isolation giving rise to only serializable schedules admits non-serializable schedules. Furthermore, it is claimed this phenomena based definition is equivalent the the locking based definition of serializable isolation which they call LOCKING SERIALIZABLE.

In fact the locking based definition, LOCKING SERIALIZABLE does not allow predicate write locks to be released until commit or abort time, thus disallowing the schedule of Example 2.

We conclude that Berenson et. al [2] phenomenon based definition of SERIALIZABLE isolation admits non-serializable schedules and further that it is not equivalent to the locking based definition given which does only admits serializable schedules.

The phenomenon **P3** does characterise the behavior in Example 1 but it is too strict. For example, the history $r_1[P]\, w2[insert\ d\ in\ P]\, a1\, c2$ is disallowed, which is serializable. We need only disallow the case when t_i commits. This produces phenomena **NP3R** below. We also require phenomenon **NP3L** to exclude behavior of the type in Example 2.

NP3R : $r_i[P] \prec w_j[d\ in\ P] \prec c_i$

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

The dirty read phenomenon characterised by **NP1** has a predicate equivalent, which we give below.

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

Again the same recoverability and database constraint arguments for phenomenon **P0** offered by Berenson et al. [2] can be used in our enriched model, and we include a predicate version of **P0** for completeness.

$\mathbf{NP2\frac{1}{4}}$: $w_i[d\ in\ P] \prec w_j[d\ in\ P] \prec e_i$

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

Isolation Level	P0 $\mathbf{NP2\frac{1}{4}}$	NP1	NP2R NP2L	NP3R NP3L $\mathbf{NP2\frac{1}{2}}$
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.

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.