Appears in the *Proceedings of the IEEE International Conference on Data Engineering*, San Diego, CA, March 2000

Generalized Isolation Level Definitions

Atul Adya
Microsoft Research,
1 Microsoft Way,
Redmond, WA 98007
adya@microsoft.com

Barbara Liskov
Laboratory for Computer Science,
Massachusetts Inst. of Technology,
Cambridge, MA 02139
liskov@lcs.mit.edu

Patrick O'Neil
Univ. of Massachusetts,
Boston, MA 02125-3393
poneil@cs.umb.edu

Abstract

Commercial databases support different isolation levels to allow programmers to trade off consistency for a potential gain in performance. The isolation levels are defined in the current ANSI standard, but the definitions are ambiguous and revised definitions proposed to correct the problem are too constrained since they allow only pessimistic (locking) implementations. This paper presents new specifications for the ANSI levels. Our specifications are portable*; they apply not only to locking implementations, but also to optimistic and multi-version concurrency control schemes. Furthermore, unlike earlier definitions, our new specifications handle predicates in a correct and flexible manner at all levels.*

1. Introduction

This paper gives new, precise definitions of the ANSI-SQL isolation levels [6]. Unlike previous proposals [13, 6, 8], the new definitions are both correct (they rule out all bad histories) and implementation-independent. Our specifications allow a wide range of concurrency control techniques, including locking, optimistic techniques [20, 2, 5], and multi-version mechanisms [9, 24]. Thus, they meet the goals of ANSI-SQL and could be used as an isolation standard.

The concept of isolation levels was first introduced in [13] under the name *Degrees of Consistency*. The goal of this work was to provide improved concurrency for workloads by sacrificing the guarantees of perfect isolation. The work in [13] and some refinements suggested by [11] set the stage for the ANSI/ISO SQL-92 definitions for isolation levels [6], where the goal was to develop a standard that was implementation-independent. However, a subsequent paper [8] showed that the definitions provided in [6] were ambiguous. That paper proposed different definitions that avoided the ambiguity problems, but, as stated in [8], these definitions were simply "disguised versions of locking" and therefore disallow optimistic and multi-version mechanisms. Thus, these definitions fail to meet the goals of ANSI-SQL with respect to implementation-independence.

Thus, we have a problem: the standard is intended to be implementation-independent, but lacks a precise definition that meets this goal. Implementation-independence is important since it provides flexibility to implementors, which can lead to better performance. Optimism can outperform locking in some environments, such as large scale, wide-area distributed systems [2, 15]; optimistic mechanisms are the schemes of choice for mobile environments; and Gemstone [22] and Oracle [24] provide serializability and Snapshot Isolation, respectively, using multi-version optimistic implementations. It is undesirable for the ANSI standard to rule out these implementations. For example, Gemstone provides serializability even though it does not meet the locking-based rules given in [8].

This paper presents new implementation-independent specifications that correct the problems with the existing definitions. Our definitions cover the weaker isolation levels that are in everyday use: Most database vendors and database programmers take advantage of levels below serializability levels to achieve better performance; in fact, READ COMMITTED is the default for some database products and database vendors recommend using this level instead of serializability if high performance is desired. Our definitions also enable database vendors to develop innovative implementations of the different levels using a wide variety of concurrency control mechanisms including locking, optimistic and multi-version mechanisms. Furthermore, our specifications handle predicate-based operations correctly

Research of A. Adya and B. Liskov was supported in part by the ARPA of the Department of Defense under contract DABT63-95-C-0005, monitored by Fort Huachuca US Army Intelligence Center, and by NSF under Grant IIS-98-02066. This research was done when Atul Adya was at MIT. Research of P. O'Neil was supported by the NSF under Grant IRI-97-11374.

at all isolation levels.

Thus, the paper makes the following contributions:

- It specifies the existing ANSI isolation levels in an implementation-independent manner. The definitions are correct (they rule out all bad histories). They are also complete (they allow all good histories) for serializability; in particular, they provide conflict-serializability [9]. It is difficult to prove completeness for lower isolation levels, but we can easily show that our definitions are more permissive than those given in [8].
- Our specifications also handle predicates correctly in a flexible manner; earlier definitions were either lock-based or incomplete [8].

Our approach can be used to define additional levels as well, including commercial levels such as Cursor Stability [11], and Oracle's Snapshot Isolation and Read Consistency [24], and new levels; for example, we have developed an additional isolation level called PL-2+, which is the weakest level that guarantees consistent reads and causal consistency with respect to transactions. Details can be found in [1].

Our definitions are given using a combination of constraints on transaction histories and graphs; we proscribe different types of cycles in a serialization graph at each isolation level. Our graphs are similar to those that have been used before for specifying serializability [9, 19, 14], semantics-based correctness criterion [4], and for defining extended transaction models [10]. Our approach is the first that applies these techniques to defining ANSI and commercial isolation levels. Our specifications are different from the multi-version theory presented in [9] since that work only describes conditions for serializability whereas we specify all ANSI/SQL-92 and other commercial isolation levels for multi-version systems. Furthermore, unlike our specifications, their definitions do not take predicates into account. Our work is also substantially different from the definitions presented in [8] since our specifications handle multi-version systems, optimistic systems and also deal with predicates in a correct and flexible manner at all isolation levels.

Relaxed correctness conditions based on semantics and extended transaction models have been suggested in the past [10, 4, 17, 7]. By contrast, our work focuses on specifying existing ANSI and commercial isolation levels that are being used by large numbers of application programmers.

The rest of this paper is organized as follows. Section 2 discusses prior work in more detail. Section 3 shows that the current definitions are inadequate and motivates the need for our work. Section 4 describes our database model. Section 5 provides our definitions for the existing ANSI isolation levels. We close in Section 6 with a discussion of what we have accomplished.

2. Previous Work

The original proposal for isolation levels [13] introduced four degrees of consistency, degrees 0, 1, 2 and 3, where degree 3 was the same as serializability. That paper, however, was concerned with locking schemes, and as a consequence the definitions were not implementation-independent.

However, that work, together with the refinement of the levels provided by Date [11], formed the basis for the ANSI/ISO SQL-92 isolation level definitions [6]. The ANSI standard had implementation-independence as a goal and the definitions were supposed to be less constraining than earlier ones. The approach taken was to proscribe certain types of bad behavior called *phenomena*; more restrictive consistency levels disallow more phenomena and serializability does not permit any phenomenon. The isolation levels were named READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE; some of these levels were intended to correspond to the degrees of [13].

The work in [8] analyzed the ANSI-SQL standard and demonstrated several problems in its isolation level definitions: some phenomena were ambiguous, while others were missing entirely. It then provided new definitions. As with the ANSI-SQL standard, various isolation levels are defined by having them disallow various phenomena. The phenomena proposed by [8] are:

P0: $w_1[x] \ldots w_2[x] \ldots (c_1 \text{ or } a_1)$
P1: $w_1[x] \ldots r_2[x] \ldots (c_1 \text{ or } a_1)$
P2: $r_1[x] \ldots w_2[x] \ldots (c_1 \text{ or } a_1)$
P3: $r_1[P] \ldots w_2[y \text{ in } P] \ldots (c_1 \text{ or } a_1)$

Proscribing P0 (which was missing in the ANSI-SQL definitions) requires that a transaction T_2 cannot write an object x if an uncommitted transaction T_1 has already modified x. This is simply a disguised locking definition, requiring T_1 and T_2 to acquire long write-locks. (Long-term locks are held until the transaction taking them commits; short-term locks are released immediately after the transaction completes the read or write that triggered the lock attempt.) Similarly, proscribing P1 requires T_1 to acquire a long write-lock and T_2 to acquire (at least) a short-term read-lock, and proscribing P2 requires the use of long read and write locks.

Phenomenon P3 deals with the queries based on predicates. Proscribing P3 requires that a transaction T_2 cannot modify a predicate P by inserting, updating, or deleting a row such that its modification changes the result of a query executed by an uncommitted transaction T_1 based on predicate P; to avoid this situation, T_1 acquires a long phantom read-lock [14] on predicate P.

Thus, these definitions only allow histories that would occur in a system using long/short read/write item/predicate locks. Since locking serializes transactions by preventing certain situations (e.g., two concurrent transactions both

Locking Isolation Level	Proscribed Phenomena	Read Locks on Data Items and Phantoms (same unless noted)	Write Locks on Data Items and Phantoms (always the same)
Degree 0	none	none	Short write locks
Degree 1 = Locking READ UNCOMMITTED	P0	none	Long write locks
Degree 2 = Locking READ COMMITTED	P0, P1	Short read locks	Long write locks
Locking REPEATABLE READ	P0, P1, P2	Long data-item read locks, Short phantom read locks	Long write locks
Degree 3 = Locking SERIALIZABLE	P0, P1, P2, P3	Long read locks	Long write locks

Figure 1. Consistency Levels and Locking ANSI-92 Isolation Levels

modifying the same object), we refer to this approach as the *preventative* approach.

Figure 1 summarizes the isolation levels as defined in [8] and relates them to a lock-based implementation. Thus the READ UNCOMMITTED level proscribes P0; READ COMMITTED proscribes P0 and P1; the REPEATABLE READ level proscribes P0 - P2; and SERIALIZABLE proscribes P0 - P3.

3. Restrictiveness of Preventative Approach

We now show that the preventative approach is overly restrictive since it rules out optimistic and multi-version implementations. As mentioned, this approach disallows all histories that would not occur in a locking scheme and *prevents* conflicting operations from executing concurrently.

The authors in [8] wanted to ensure that multi-object constraints (e.g., constraints like $x + y = 10$) are not observed as violated by transactions that request an isolation level such as serializability. They showed that histories such as H_1 and H_2 are allowed by one interpretation of the ANSI standard (at the SERIALIZABLE isolation level) even though they are non-serializable:

H_1: $r_1(x, 5)$ $w_1(x, 1)$ $r_2(x, 1)$ $r_2(y, 5)$ c_2 $r_1(y, 5)$ $w_1(y, 9)$ c_1

H_2: $r_2(x, 5)$ $r_1(x, 5)$ $w_1(x, 1)$ $r_1(y, 5)$ $w_1(y, 9)$ c_1 $r_2(y, 9)$ c_2

In both cases, T_2 observes an inconsistent state (it observes invariant $x + y = 10$ to be violated). These histories are not allowed by the preventative approach; H_1 is ruled out by P1 and H_2 is ruled out by P2.

Optimistic and multi-version mechanisms [2, 5, 9, 20, 22] that provide serializability also disallow non-serializable histories such as H_1 and H_2. However, they allow many legal histories that are not permitted by P0, P1, P2, and P3. Thus, the preventative approach disallows such implementations. Furthermore, it rules out histories that really occur in practical implementations.

Phenomenon P0 can occur in optimistic implementations since there can be many uncommitted transactions modifying local copies of the same object concurrently; if necessary, some of them will be forced to abort so that serializability can be provided. Thus, disallowing P0 can rule out optimistic implementations.

Condition P1 precludes transactions from reading updates by uncommitted transactions. Such reads are disallowed by many optimistic schemes, but they are desirable in mobile environments, where commits may take a long time if clients are disconnected from the servers [12, 16]; furthermore, reads from uncommitted transactions may be desirable in high traffic hotspots [23]. For example, in history H_1, if T_2 reads T_1's values for both *x* and *y*, it can be serialized after T_1:

$H_{1'}$: $r_1(x, 5)$ $w_1(x, 1)$ $r_1(y, 5)$ $w_1(y, 9)$ $r_2(x, 1)$ $r_2(y, 9)$ c_1 c_2

The above history can occur in a mobile system, but P1 disallows it. In such a system, commits can be assumed to have happened "tentatively" at client machines [12, 16]; later transactions may observe modifications of those tentative transactions. When the client reconnects with the servers, its work is checked to determine if consistency has been violated and the relevant transactions are aborted. Of course, if dirty reads are allowed, cascading aborts can occur, e.g., in history $H_{1'}$, T_2 must abort if T_1 aborts; this problem can be alleviated by using compensating actions [18, 26, 19].

Proscribing phenomenon P2 disallows a modification to an object that has been read by an uncommitted transaction (P3 rules out a similar situation with respect to predicates). As with P0, uncommitted transactions may read/write the same object concurrently in an optimistic implementation. There is no harm in allowing phenomenon P2 if transactions commit in the right order. For example, in history H_2 given above, if T_2 reads the old values of *x* and *y*, the transactions can be serialized in the order T_2; T_1:

$H_{2'}$: $r_2(x, 5)$ $r_1(x, 5)$ $w_1(x, 1)$ $r_1(y, 5)$ $r_2(y, 5)$ $w_1(y, 9)$ c_2 c_1

The real problem with the preventative approach is that the phenomena are expressed in terms of single-object histories. However, the properties of interest are often multi-object constraints. To avoid problems with such constraints, the phenomena need to restrict what can be done with individual objects more than is necessary. Our approach avoids this difficulty by using specifications that capture constraints on multiple objects directly. Furthermore, the definitions in the preventative approach are not applicable to multi-version systems since they are described in terms of objects rather than in terms of versions. On the other hand, our specifications deal with multi-version and single-version histories.

The approach in [8] only allows schemes that provide the same guarantees for running and committed transactions (a lock-based implementation does indeed have this

property). However, many optimistic mechanisms provide weak guarantees to transactions as they run while providing strong guarantees such as serializability for committed transactions. Our definitions allow different isolation guarantees for committed and running transactions; in this paper, we only present guarantees for committed transactions.

4. Database Model and Transaction Histories

We now describe our database model, transaction histories, and serialization graphs. We use a multi-version model similar to the one presented in [9]. However, unlike [9], our model incorporates predicates also. Furthermore, we allow predicate behavior that is possible in non-locking based systems.

4.1. Database Model

The database consists of objects that can be read or written by transactions; in a relational database system, each row or tuple is an object. Each transaction reads and writes objects and indicates a total order in which these operations occur.

An object has one or more versions. However, transactions interact with the database only in terms of objects; the system maps each operation on an object to a specific version of that object. A transaction may read versions created by committed, uncommitted, or even aborted transactions; constraints imposed by some isolation levels will prevent certain types of reads, e.g., reading versions created by aborted transactions.

When a transaction writes an object *x*, it creates a new version of *x*. A transaction T_i can modify an object multiple times; its first modification of object *x* is denoted by $x_{i.1}$, the second by $x_{i.2}$, and so on. Version x_i denotes the final modification of *x* performed by T_i before it commits or aborts. A transaction's last operation, *commit* or *abort*, indicates whether its execution was successful or not; there is at most one commit or abort operation for each transaction.

The *committed state* reflects the modifications of committed transactions. When transaction T_i commits, each version x_i created by T_i becomes a part of the committed state and we say that T_i *installs* x_i; the system determines the ordering of x_i relative to other committed versions of x. If T_i aborts, x_i does not become part of the committed state.

Conceptually, the initial committed state comes into existence as a result of running a special initialization transaction, T_{init}. Transaction T_{init} creates all objects that will ever exist in the database; at this point, each object *x* has an initial version, x_{init}, called the *unborn* version. When an application transaction creates an object *x* (e.g., by inserting a tuple in a relation), we model it as the creation of a *visible* version for *x*. Thus, a transaction that loads the database creates the initial visible versions of the objects being inserted. When a transaction T_i deletes an object *x* (e.g., by deleting a tuple from some relation), we model it as the creation of a special *dead* version, i.e., in this case, x_i is a dead version. Thus, object versions can be of three kinds — unborn, visible, and dead; the ordering relationship between these versions is discussed in Section 4.2.

If an object *x* is deleted from the committed database state and inserted later, we consider the two incarnations of *x* to be distinct objects. When a transaction T_i performs an insert operation, the system selects a *unique* object *x* that has never been selected for insertion before and T_i creates a visible version of *x* if it commits.

We assume object versions exist forever in the committed state to simplify the handling of inserts and deletes, i.e., we simply treat inserts/deletes as write (update) operations. An implementation only needs to maintain visible versions of objects, and a single-version implementation can maintain just one visible version at a time. Furthermore, application transactions in a real system access only visible versions.

4.2. Transaction Histories

We capture what happens in an execution of a database system by a history. A *history H* over a set of transactions consists of two parts — a partial order of events *E* that reflects the operations (e.g., read, write, abort, commit) of those transactions, and a version order, $\ll$, that is a total order on committed versions of each object.

Each event in a history corresponds to an operation of some transaction, i.e., read, write, commit, or abort. A write operation on object *x* by transaction T_i is denoted by $w_i(x_i)$ (or $w_i(x_{i.m})$); if it is useful to indicate the value *v* being written into x_i, we use the notation, $w_i(x_i, v)$. When a transaction T_j reads a version of *x* that was created by T_i, we denote this as $r_j(x_i)$ (or $r_j(x_{i.a})$). If it is useful to indicate the value *v* being read, we use the notation $r_j(x_i, v)$.

The partial order of events *E* in a history obeys the following constraints:

- It preserves the order of all events within a transaction including the commit and abort events.
- If an event $r_j(x_{i.m})$ exists in *E*, it is preceded by $w_i(x_{i.m})$ in *E*, i.e., a transaction T_j cannot read version $x_{i.m}$ of object *x* before it has been produced by T_i. Note that the version read by T_j is not necessarily the most recently installed version in the committed database state; also, T_i may be uncommitted when $r_j(x_{i.m})$ occurs.
- If an event $w_i(x_{i.m})$ is followed by $r_i(x_j)$ without an intervening event $w_i(x_{i.n})$ in *E*, x_j must be $x_{i.m}$. This condition ensures that if a transaction modifies object *x* and later reads *x*, it will observe its last update to *x*.

- The history must be *complete*: if E contains a read or write event that mentions a transaction T_i, E must contains a commit or abort event for T_i.

A history that is not complete can be completed by appending abort events for uncommitted transactions in E. Adding these events is intuitively correct since any implementation that allows a commit of a transaction that reads from an uncommitted transaction T_i can do so only if it is legal for T_i to abort later.

For convenience, we will present event histories in examples as a total order (from left to right) that is consistent with the partial order.

The second part of a history H is the version order, $\ll$, that specifies a total order on versions of each object created by *committed* transactions in H; there is no ordering of versions due to uncommitted or aborted transactions. We also refer to versions due to committed transactions in H as *committed versions*. We impose two constraints on a history's version order for different kinds of committed versions:

- the version order of each object *x* contains exactly one initial version, x_{init}, and at most one committed dead version, x_{dead}.
- x_{init} is *x*'s first version in its version order and x_{dead} is its last version (if it exists); all committed visible versions are placed between x_{init} and x_{dead}.

We additionally constrain the system to allow reads only of visible versions:

- if $r_j(x_i)$ occurs in a history, then x_i is a visible version.

For convenience, we will only show the version order for visible versions in our example histories; in cases where unborn or dead versions help in illustrating an issue, we will show some of these versions as well.

The version order in a history H can be different from the order of write or commit events in H. This flexibility is needed to allow certain optimistic and multi-version implementations where it is possible that a version x_i is placed before version x_j in the version order ($x_i \ll x_j$) even though x_i is installed in the committed state *after* version x_j was installed. For example, in history $H_{write-order}$,

$$H_{write-order}\text{: } w_1(x_1)\ w_2(x_2)\ w_2(y_2)\ c_1\ c_2 \\ r_3(x_1)\ w_3(x_3)\ w_4(y_4)\ a_4 \qquad [x_2 \ll x_1]$$

the database system has chosen the version order $x_2 \ll x_1$ even though T_1 commits before T_2. Note that there are no constraints on x_3 (yet) or y_4 since these versions correspond to uncommitted and aborted transactions, respectively. Note also that the naming of transactions does not indicate their commit order, e.g., in history $H_{write-order}$, T_2 is serialized before T_1.

4.3. Predicates

We now discuss how predicates are handled in our model. We assume that predicates are used with relations in a relational database system. There are three types of modification operations on relations: updates, inserts and deletes; inserts and deletes change the number of tuples in a relation.

In our model, the database is divided into relations and each tuple (and all its versions) exists in some relation. As before, unborn and dead versions exist for a tuple before the tuple's insertion and after its deletion. An important point to note here is that a tuple's relation is known in our model when the database is initialized by T_{init}, i.e., *before* the tuple is inserted by an application transaction. Of course, this assumption is needed only at a conceptual level. In an implementation, the system need not know the relation of all tuples that will be created in the system; it just needs to know a tuple *x*'s relation when *x* is inserted in the database.

A predicate P identifies a Boolean condition (e.g., as in the WHERE clause of a SQL statement) and the relations on which the condition has to be applied; one or more relations can be specified in P. All tuples that match this condition are read or modified depending on whether a predicate-based read or write is being considered.

Definition 1 : Version set of a predicate-based operation. When a transaction executes a read or write based on a predicate P, the system selects a version for *each* tuple in P's relations. The set of selected versions is called the *Version set* of this predicate-based operation and is denoted by Vset(P).

The version set defines the state that is observed to evaluate a predicate P; as discussed later, P's Boolean condition is applied on the versions in Vset(P) to determine which tuples satisfy P. Since we select a version for all possible tuples in P's relations, this set will be very large (it includes unborn and possibly dead versions of some tuples). For convenience, in our examples we will only show visible versions in a version set; to better explain some examples, we will sometimes also show some unborn and dead versions.

Our approach of observing some version of each tuple allows us to handle the phantom problem [14] in a simple manner. Of course, this does not constrain implementations to perform these observations; e.g., an implementation could use an index.

4.3.1 Predicate-based Reads

If a transaction T_i performs reads based on a predicate P (e.g., in a SQL statement), the system (conceptually) accesses all versions in Vset(P). Then, the system determines which tuples *match* predicate P by evaluating P's Boolean condition on the versions in Vset(P); tuples whose unborn

and dead versions were selected in the previous step do not match. If the system reads the matched versions as part of the query, these reads show up as *separate* events in the history. Thus, a query based on a predicate P by T_i is represented in a history as r_i(P: Vset(P)) $r_i(x_j)$ $r_i(y_k)$ ..., where x_j, y_k are the versions in Vset(P) that match P, and T_i reads these versions. If T_i does not read the matched objects, the events $r_i(x_j)$ and $r_i(y_k)$ do not show up in the history, e.g., T_i could simply use the count of tuples that matched P.

For example, suppose transaction T_i executes the following SQL query:

SELECT * FROM EMPLOYEE WHERE DEPT = SALES;

This query (conceptually) accesses a version of every visible tuple in the Employee relation (e.g., x_1 and y_2) and the unborn/dead versions of other tuples in this relation (e.g., z_{init}). Suppose that version x_1 matches the predicate and y_2 does not match; recall that unborn versions such as z_{init} cannot match the predicate. This predicate-based read could be shown in a history as r_i(Dept=Sales: x_1; y_2) $r_i(x_1)$; here, we do not show unborn or dead versions in the version set. Note that the read of x_1 shows up as a separate event in the history; if T_i had just determined the number of tuples matching the predicate (using SELECT COUNT), the event $r_i(x_1)$ would not have been included.Thus, the history only shows reads of versions that were actually observed by T_i.

4.3.2 Predicate-based Modifications

A modification based on a predicate P is modeled as a predicate-based read followed by write operations on tuples that match P. (Although this approach is weaker than the one used in [1], it models the behavior of commercial databases.) For example, suppose transaction T_i executes the following code for the employee database discussed above:

UPDATE EMPLOYEE SAL = SAL + $10 WHERE DEPT=SALES;

Suppose that the system selects versions, x_1, y_2, and z_{init} for this operation. If x_1 matches the predicate but y_2 and z_{init} do not, the following events are added to the history: r_i(Dept=Sales: x_1; y_2) $w_i(x_i)$.

If the predicate-based write deletes objects, dead versions are installed for all the matching tuples (i.e., these tuples are deleted). Thus, if a transaction T_i deletes all employees from the Sales department in the above scenario, the following events are added to the history: r_i(Dept=Sales: x_1; y_2) w_i(x_i, dead). Note that the events for deletes and updates are similar. However, there is a difference: in the deletion example, x_i is a dead version (for illustrative purposes, we have shown the value "dead" being put in x_i) and cannot be used further whereas in the update case, x_i can be used later.

Inserts are handled in a similar manner. For example, consider the following statement that copies employees whose commission exceeds 25% of their salary into the BONUS table (this statement is executed by transaction T_1):

T_1: INSERT INTO BONUS SELECT NAME, SAL, COMM
FROM EMP WHERE COMM > 0.25 * SAL;

Here is a possible history for T_1's execution in our model:

H_{insert}: r_1(comm > 0.25 * sal: x_0, z_0) $r_1(x_0)$ $w_1(y_1)$ c_1

In this history, x_0 matches the predicate-based query; therefore it is read by T_1 to generate tuple y_1 that is inserted into the Bonus table.

4.4. Conflicts and Serialization Graphs

We first define the different types of read/write conflicts that can occur in a database system and then use them to specify serialization graphs. We define three kinds of *direct conflicts* that capture conflicts of two different committed transactions on the same object or intersecting predicates. For convenience, we have separated the definitions of predicate-based conflicts and regular conflicts.

4.4.1 Read Dependencies

Read dependencies occur when one transaction reads a relevant version produced by some other transaction. We use the following definition for specifying read-dependencies:

Definition 2 : Change the Matches of a Predicate-Based Read. We say that a transaction T_i *changes the matches* of a predicate-based read r_j(P: Vset(P)) if T_i installs x_i, x_h immediately precedes x_i in the version order, and x_h matches P whereas x_i does not or vice-versa. In this case, we also say that x_i changes the matches of the predicate-based read.

The above definition identifies T_i to be a transaction where a change occurs for the matched set of r_j(P: Vset(P)).

Definition 3 : Directly Read-Depends. We say that T_j *directly read-depends* on transaction T_i if it directly item-read-depends or directly predicate-read-depends on T_i.

Directly item-read-depends: We say that T_j *directly item-read-depends* on T_i if T_i installs some object version x_i and T_j reads x_i.

Directly predicate-read-depends: Transaction T_j *directly predicate-read-depends* on T_i if T_j performs an operation r_j(P: Vset(P)), $x_k \in$ Vset(P), $i = k$ or $x_i \ll x_k$, and x_i changes the matches of r_j(P: Vset(P)).

If T_j performs a predicate-based read r_j(P: Vset(P)), it read depends on T_i if T_i performs a write that is "relevant" to T_j's read, i.e., T_i is a transaction before T_j that changed the matches of T_j's read. Note that all tuples in the version set of a predicate-based read are considered to be accessed, including tuples that do not match the predicate. The versions that are actually read by transaction T_j show up as normal

read events. Other versions in the version set are essentially *ghost reads*, i.e., their values are not observed by the predicate-based read but read-dependencies are established for them as well.

The rule for predicate-read-dependencies captures the idea that what matters for a predicate is the set of tuples that match or do not match and not their values. Furthermore, of all the transactions that have caused the tuples to match (or not match) for r_j(P: Vset(P)), we use the latest transaction where a change to Vset(P) occurs rather than using the latest transaction that installed the versions in Vset(P). This rule ensures that we capture the minimum possible conflicts for predicate-based reads. For example, consider the history:

$H_{pred-read}$: $w_0(x_0)$ c_0 $w_1(x_1)$ c_1 $w_2(x_2)$
r_3(Dept=Sales: x_2, y_0) $w_2(y_2)$ c_2 c_3 $[x_0 \ll x_1 \ll x_2, y_0 \ll y_2]$

Here, transaction T_0 inserts object *x* in the Sales department, T_1 changes *x*'s department to Legal, and T_2 changes the phone number of *x* but not its department. Transaction T_3 selects all employees in the Sales department. In this case, even though T_3's version set contains x_2, we add a predicate-read-dependency from T_1 to T_3 because T_2's update of *x* is irrelevant for T_3's read. Note that this history is serializable in the order T_0, T_1, T_3, T_2.

4.4.2 Anti-Dependencies

An anti-dependency occurs when a transaction overwrites a version observed by some other transaction.

To define anti-dependencies, it is useful to define what it means to overwrite a predicate-based operation.

Definition 4: Overwriting a predicate-based read. We say that a transaction T_j *overwrites* an operation r_i(P: Vset(P)) if T_j installs x_j such that $x_k \in$ Vset(P), $x_k \ll x_j$, and x_j changes the matches of r_i(P: Vset(P)).

Now we can define anti-dependencies.

Definition 5: Directly Anti-Depends. Transaction T_j *directly anti-depends* on transaction T_i if it directly item-anti-depends or directly predicate-anti-depends on T_i.

Directly item-anti-depends: We say that T_j *directly item-anti-depends* on transaction T_i if T_i reads some object version x_k and T_j installs *x*'s next version (after x_k) in the version order. Note that the transaction that wrote the later version directly item-anti-depends on the transaction that read the earlier version.

Directly predicate-anti-depends: We say that T_j *directly predicate-anti-depends* on T_i if T_j overwrites an operation r_i(P: Vset(P)), i.e., T_j installs a later version of some object that changes the matches of a predicate-based read performed by T_i.

Read-dependencies and anti-dependencies are treated similarly for predicates, i.e., we add an edge whenever a predicate's matched set is changed. The difference between item-anti-depends and predicate-anti-depends is also similar. For item-anti-depends, the overwriting transaction must produce the very next version of the read object, while for predicate-anti-depends it simply produces a later version that changes the matched tuples of the predicate.

The definition for predicate-anti-depends handles inserts and deletes. For example, consider the employee database scenario described in Section 4.3 that contains visible versions of two tuples *x* and *y*. Suppose T_i executes a query that selects all Employees in the Sales department, and the query's version set contains versions x_1 and y_2 (along with unborn/dead versions of other tuples), and x_1 is in Sales and y_2 is not. A later transaction T_j will directly predicate-anti-depend on T_i if T_j adds a new employee to the Sales department, moves *y* to Sales, removes *x* from Sales, or deletes *x* from the database.

In a two-phase locking implementation (for providing serializability), if a transaction T_1 performs a read based on predicate P and T_2 tries to insert an object *x* covered by P's predicate lock, T_2 is delayed till T_1 finishes. In our model, T_1 reads x_{init} and T_2 creates a later version x_2. If T_2 changes the matches by T_1's read, T_2 predicate-anti-depends on T_1. Note that T_1's predicate read-locks delay T_2 even if T_2 does not change the objects matched by P. Our definitions are more flexible and permit implementations that allow T_2 to proceed in such cases, e.g., precision locks and granular locks [14].

4.4.3 Write Dependencies

Write dependencies occur when one transaction overwrites a version written by another transaction.

Definition 6: Directly Write-Depends. A transaction T_j *directly write-depends* on T_i if T_i installs a version x_i and T_j installs *x*'s next version (after x_i) in the version order.

Note that there is no notion of predicate-write-depends since predicate-based modifications are modeled as queries followed by writes on individual tuples.

4.4.4 Serialization Graphs

Now we can define the Direct Serialization Graph or DSG. This graph is called "direct" since it is based on the direct conflicts discussed above. In the graph we will denote direct read-dependencies by $T_i \xrightarrow{wr} T_j$, direct write-dependencies by $T_i \xrightarrow{ww} T_j$, and direct anti-dependencies by $T_i \xrightarrow{rw} T_j$. Figure 2 summarizes this notation and reviews the definitions for direct dependencies.

Conflicts Name	*Description* (T_j conflicts on T_i)	Notation in DSG
Directly write-depends	T_i installs x_i and T_j installs *x*'s next version	$T_i \xrightarrow{ww} T_j$
Directly read-depends	T_i installs x_i, T_j reads x_i or T_j performs a predicate-based read, x_i changes the matches of T_j's read, and x_i is the same or an earlier version of *x* in T_j's read	$T_i \xrightarrow{wr} T_j$
Directly anti-depends	T_i reads x_h and T_j installs *x*'s next version or T_i performs a predicate-based read and T_j overwrites this read	$T_i \overset{rw}{- -\rightarrow} T_j$

Figure 2. Definitions of direct conflicts between transactions.

Definition 7 : Direct Serialization Graph.

We define the direct serialization graph arising from a history H, denoted by DSG(H), as follows. Each node in the graph corresponds to a *committed* transaction and directed edges correspond to different types of direct conflicts. There is a *read/write/anti-dependency* edge from transaction T_i to transaction T_j if T_j *directly read/write/anti-depends* on T_i.

A DSG does not capture all information in a history and hence it does not replace the history, e.g., a DSG only records information about committed transactions. The history is still available if needed, and in fact, we use the history instead of the DSG for some conditions.

As an example, consider the following history:

H_{serial}: $w_1(z_1)$ $w_1(x_1)$ $w_1(y_1)$ $w_3(x_3)$ c_1 $r_2(x_1)$ $w_2(y_2)$
c_2 $r_3(y_2)$ $w_3(z_3)$ c_3 $[x_1 \ll x_3, y_1 \ll y_2, z_1 \ll z_3]$

Figure 3 shows the DSG for this history. As we can see, these transactions are serializable in the order T_1; T_2; T_3.

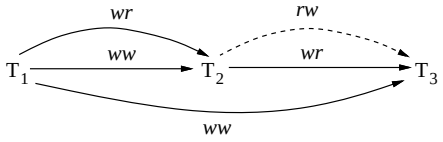

Figure 3. DSG for history H_{serial}

It is also useful to have additional dependency relations:

Definition 8 : Depends. A transaction T_j *directly depends* on T_i if T_j directly write-depends or directly read-depends on T_i. We say that T_j *depends* on T_i in H if there is a path from T_i to T_j in DSG(H) consisting of one or more dependency edges.

5. New Generalized Isolation Specifications

We now present our specifications for the existing ANSI isolation levels. We developed our conditions by studying the motivation of the original definitions [13] and the problems that were addressed by the phenomena in [8]. This enabled us to develop implementation-independent specifications that capture the essence of the ANSI definitions, i.e., we disallow undesirable situations while allowing histories that are permitted by a variety of implementations.

Like the previous approaches, we will define each isolation level in terms of phenomena that must be avoided at each level. Our phenomena are prefixed by "G" to denote the fact that they are general enough to allow locking and optimistic implementations; these phenomena are named G0, G1, and so on (by analogy with P0, P1, etc of [6]). We will refer to the new levels as PL levels (where PL stands for "portable level") to avoid the possible confusion with the degrees of isolation given in [8, 13].

5.1. Isolation Level PL-1

Disallowing phenomenon P0 ensures that writes performed by T_1 are not overwritten by T_2 while T_1 is still uncommitted. There seem to be two reasons why this proscription might be desirable:

1. It simplifies recovery from aborts. In the absence of this proscription, a system that allows writes to happen in place cannot recover the pre-states of aborted transactions using a simple undo log approach. For example, suppose T_1 updates *x* (i.e., $w_1(x_1)$), T_2 overwrites *x*, and then T_1 aborts. The system must not restore *x* to T_1's pre-state. However, if T_2 aborts later, *x* must be restored to T_1's pre-state and not to x_1.
2. It serializes transactions based on their writes alone. For example, if T_2 updates an object *x* and T_1 overwrites *x*, there should not be another object *y* in which the reverse occurs, i.e., all writes of T_2 must be ordered before or after all writes of T_1.

The first reason does not seem relevant to all systems. Instead, it is based on a particular implementation of recovery, and other implementations are possible. For example, the Thor system [21] maintains temporary versions of objects for an uncommitted transaction T_i and discards these versions if T_i aborts.

Serializing transactions based on writes is a useful property since it ensures that updates of conflicting transactions

are not interleaved. This property is captured by phenomenon G0 and we define **PL-1** as the level in which G0 is disallowed:

> **G0: Write Cycles.** A history H exhibits phenomenon G0 if DSG(H) contains a directed cycle consisting entirely of write-dependency edges.

For example, history H_{wcycle}

H_{wcycle}: $w_1(x_1, 2)\ w_2(x_2, 5)\ w_2(y_2, 5)\ c_2\ \ w_1(y_1, 8)\ c_1$
$[x_1 \ll x_2, y_2 \ll y_1]$

is disallowed by PL-1 because the updates on *x* and *y* occur in opposite orders, causing a cycle in the graph. Figure 4 shows the DSG for this history.

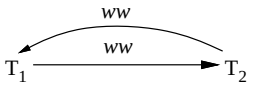

Figure 4. DSG for history H_{wcycle}

Our PL-1 specification is more permissive than Degree 1 of [8] since G0 allows concurrent transactions to modify the same object whereas P0 does not. Thus, non-serializable interleaving of write operations is possible among uncommitted transactions as long as such interleavings are disallowed among committed transactions (e.g., by aborting some transactions).

The lock-based implementation of PL-1 (long write-locks) disallows G0 since two concurrent transactions, T_i and T_j, cannot modify the same object; therefore, all writes of T_j either precede or follow all writes of T_i.

Note that since predicate-based modifications are modeled as queries followed by normal writes, PL-1 provides weak guarantees for such updates. For example, consider the following history in which transaction T_2 increments the salaries of all employees for which "Dept = Sales", and T_1 adds two employees, *x* and *y*, to the Sales department.

$H_{pred-update}$: $w_1(x_1)\ \ r_2(\text{Dept=Sales: } x_1; y_{init})\ \ w_1(y_1)$
$w_2(x_2)\ \ c_1\ \ c_2 \qquad [x_{init} \ll x_1 \ll x_2, y_{init} \ll y_1]$

The updates of transactions T_1 and T_2 are interleaved in this history (*x*'s salary is updated but *y*'s salary is not). This interleaving is allowed at PL-1 since there is no write-dependency cycle in the DSG (there is a write-dependency edge from T_1 to T_2 since $x_1 \ll x_2$).

5.2. Isolation Level PL-2

If a system disallows only G0, it places no constraints on reads: a transaction is allowed to read modifications made by committed, uncommitted, or even aborted transactions. Proscribing phenomenon P1 in [6] was meant to ensure that T_1 updates could not be read by T_2 while T_1 was still uncommitted. There seem to be three reasons why disallowing P1 (in addition to P0) might be useful:

1. It prevents a transaction T_2 from committing if T_2 has read the updates of a transaction that might later abort.
2. It prevents transactions from reading intermediate modifications of other transactions.
3. It serializes committed transactions based on their read/write-dependencies (but not their anti-dependencies). That is, if transaction T_2 depends on T_1, T_1 cannot depend on T_2.

Disallowing P1 (together with P0) captures all three of these issues, but does so by preventing transactions from reading or writing objects written by transactions that are still uncommitted. Instead, we address these three issues by the following three phenomena, G1a, G1b, and G1c.

> **G1a: Aborted Reads.** A history H shows phenomenon G1a if it contains an aborted transaction T_1 and a committed transaction T_2 such that T_2 has read some object (maybe via a predicate) modified by T_1. Phenomenon G1a can be represented using the following history fragments:
>
> $w_1(x_{1.i})\ \ldots\ r_2(x_{1.i})\ \ldots$ (a_1 and c_2 in any order)
> $w_1(x_{1.i})\ \ldots\ r_2(\text{P: } x_{1.i}, ...)\ \ldots$ (a_1 and c_2 in any order)

Proscribing G1a ensures that if T_2 reads from T_1 and T_1 aborts, T_2 must also abort; these aborts are also called *cascaded aborts* [9]. In a real implementation, the condition also implies that if T_2 reads from an uncommitted transaction T_1, T_2's commit must be delayed until T_1's commit has succeeded [9, 14].

> **G1b: Intermediate Reads**. A history H shows phenomenon G1b if it contains a committed transaction T_2 that has read a version of object *x* (maybe via a predicate) written by transaction T_1 that was not T_1's final modification of *x*. The following history fragments represent this phenomenon:
>
> $w_1(x_{1.i})\ \ldots\ r_2(x_{1.i})\ \ldots\ w_1(x_{1.j})\ \ldots\ c_2$
> $w_1(x_{1.i})\ \ldots\ r_2(\text{P: } x_{1.i}, ...)\ \ldots\ w_1(x_{1.j})\ \ldots\ c_2$

Proscribing G1b ensures that transactions are allowed to commit only if they have read final versions of objects created by other transactions. Note that disallowing G1a and G1b ensures that a committed transaction has read only object states that existed (or will exist) at some instant in the committed state.

> **G1c: Circular Information Flow.** A history H exhibits phenomenon G1c if DSG(H) contains a directed cycle consisting entirely of dependency edges.

Intuitively, disallowing G1c ensures that if transaction T_2 is affected by transaction T_1, it does not affect T_1, i.e., there is a unidirectional flow of information from T_1 to T_2. Note

that G1c includes G0. We could have defined a weaker version of G1c that only concerned cycles with at least one read-dependency edge, but it seemed simpler not to do this.

Phenomenon G1 captures the essence of no-dirty-reads and is comprised of G1a, G1b and G1c. We define isolation level **PL-2** as one in which phenomenon G1 is disallowed. Proscribing G1 is clearly weaker than proscribing P1 since G1 allows reads from uncommitted transactions. The lock-based implementation of PL-2 disallows G1 because the combination of long write-locks and short read-locks ensures that if T_i reads a version produced by T_j, T_j must have committed already (i.e., G1a, G1b not possible) and therefore T_j cannot read a version produced by T_i (i.e., G1c not possible).

Our PL-2 definition treats predicate-based reads like normal reads and provides no extra guarantees for them; we believe this approach is the most useful and flexible. Other approaches, such as requiring that each predicate-based operation is atomic with respect to other predicate-based operations, are discussed in [1].

5.3. Isolation Level PL-3

In a system that proscribes only G1, it is possible for a transaction to read inconsistent data and therefore to make inconsistent updates. Although disallowing phenomenon P2 prevents such situations (e.g., H_2 presented in Section 3), it also prevents legal histories such as $H_{2'}$ (which is also discussed in Section 3) and hence, disallows many optimistic and multi-version concurrency control schemes. What we need is to prevent transactions that perform inconsistent reads or writes from committing. This is accomplished by the following condition:

> **G2: Anti-dependency Cycles.** A history H exhibits phenomenon G2 if DSG(H) contains a directed cycle with one or more anti-dependency edges.

We define **PL-3** as an isolation level that proscribes G1 and G2. Thus, all cycles are precluded at this level. Of course, the lock-based implementation of PL-3 (long read/write-locks) disallows phenomenon G2 also since two-phase locking is known to provide complete serializability.

Proscribing G2 is weaker than proscribing P2, since we allow a transaction T_j to modify object *x* even after another uncommitted transaction T_i has read *x*. Our PL-3 definition allows histories such as $H_{1'}$ and $H_{2'}$ (presented in Section 3) that were disallowed by the preventative definitions.

The conditions given in [9] provides view-serializability whereas our specification for PL-3 provides conflict-serializability (this can be shown using theorems presented in [9]). All realistic implementations provide conflict-serializability; thus, our PL-3 conditions provide what is normally considered as serializability.

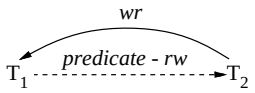

Figure 5. Direct serialization graph for history $H_{phantom}$ (T_0 is not shown)

5.4. Isolation Level PL-2.99

The level called REPEATABLE READ or Degree 2.99 in [6] provides less than full serializability with respect to predicates. In particular, it uses long locks for all operations except predicate reads for which it used short locks, i.e., it ensures serializability with respect to regular reads and provides guarantees similar to degree 2 for predicate reads. Thus, anti-dependency cycles due to predicates can occur at this level.

We define level **PL-2.99** as one that proscribes G1 and G2-item:

> **G2-item: Item Anti-dependency Cycles.** A history H exhibits phenomenon G2-item if DSG(H) contains a directed cycle having one or more item-anti-dependency edges.

For example, consider the following history:

$H_{phantom}$: r_1(Dept=Sales: x_0, 10; y_0, 10) $r_1(x_0, 10)$ $r_2(y_0, 10)$ $r_2(Sum_0, 20)$ $w_2(z_2, 10)$ $w_2(Sum_2, 30)$ c_2 $r_1(Sum_2, 30)$ c_1
$[Sum_0 \ll Sum_2, z_{init} \ll z_2]$

When T_1 performs its query, there are exactly two employees, *x* and *y*, both in Sales (we show only visible versions in the history). T_1 sums up the salaries of these employees and compares it with the sum-of-salaries maintained for this department. However, before it performs the final check, T_2 inserts a new employee, z_2, in the Sales department, updates the sum-of-salaries, and commits. Thus, when T_1 reads the new sum-of-salaries value it finds an inconsistency.

The DSG for $H_{phantom}$ is shown in Figure 5. This history is ruled out by PL-3 but permitted by PL-2.99 because the DSG contains a cycle only if predicate anti-dependency edges are considered.

5.5. Mixing of Isolation Levels

So far, we have only discussed systems in which all transactions are provided the same guarantees. However, in general, applications may run transactions at different levels and we would like to understand how these transactions interact with each other. This section discusses how we model such mixed systems.

In real database systems, each SQL statement in a transaction T_i may be executed atomically even though T_i is

Level	Phenomena disallowed	Informal Description (T_i can commit only if:)
PL-1	G0	T_i's writes are completely isolated from the writes of other transactions
PL-2	G1	T_i has only read the updates of transactions that have committed by the time T_i commits (along with PL-1 guarantees)
PL-2.99	G1, G2-item	T_i is completely isolated from other transactions with respect to data items and has PL-2 guarantees for predicate-based reads
PL-3	G1, G2	T_i is completely isolated from other transactions, i.e., all operations of T_i are before or after all operations of any other transaction

Figure 6. Summary of portable ANSI isolation levels

executed at a lower isolation level. Mixed systems in which individual SQL statements are executed atomically are discussed in [1].

In a mixed system, each transaction specifies its level when it starts and this information is maintained as part of the history and used to construct a *mixed serialization graph* or *MSG*. Like a DSG, the MSG contains nodes corresponding to committed transactions and edges corresponding to dependencies, but only dependencies relevant to a transaction's level or *obligatory* dependencies show up as edges in the graph. Transaction T_i has an obligatory conflict with transaction T_j if T_j is running at a higher level than T_i, T_i conflicts with T_j, and the conflict is relevant at T_j's level. For example, an anti-dependency edge from a PL-3 transaction to a PL-1 transaction is an obligatory edge since overwriting of reads matters at level PL-3.

Edges are added as follows: Since write-dependencies are relevant at all levels, we retain all such edges. For a PL-2 or PL-3 node T_i, since reads are important, read-dependencies coming into T_i are added. Similarly, we add all outgoing anti-dependency edges from PL-3 transactions to other nodes.

Now we can define correctness for a mixed history:

Definition 9: Mixing-Correct. A history H is mixing-correct if MSG(H) is acyclic and phenomena G1a and G1b do not occur for PL-2 and PL-3 transactions.

It is possible to restate the above definition as an analog of the Isolation Theorem [14]:

> **Mixing Theorem:** If a history is mixing-correct, each transaction is provided the guarantees that pertain to its level.

The above theorem holds at the level of a history and is independent of how synchronization is implemented [1]. Note that the guarantees provided to each level are with respect to the MSG. The reason is that an MSG considers the presence of transactions at other levels whereas a DSG is simply constructed with all edges. An MSG is useful for determining correctness if PL-1 and PL-2 transactions "know" what they are doing whereas a DSG ensures correctness without making any assumptions about the operations of lower level transactions.

A mixed system can be implemented using locking (with the standard combination of short and long read/write locks). But it can also be implemented using other techniques. For example an optimistic implementation would attempt to fit each committing transaction into the serial order based on its own requirements (for its level) and its obligations to transactions running at higher levels, and would abort the transaction if this is not possible. An optimistic implementation that is mixing-correct is presented in [1].

5.6. Discussion

We summarize the isolation levels discussed in this section in Figure 6.

These levels are defined to impose constraints only when transactions commit; they do not constrain transactions as they run, although if something bad happens (e.g., a PL-3 transactions observes an inconsistency), they do force aborts. Analogs of the levels that constrain executing transactions are given in [1]; these definitions use slightly different graphs, containing nodes for committed transactions plus a node for the executing transaction.

6. Conclusions

This paper has presented new, precise specifications of the ANSI-SQL isolation levels. Unlike previous proposals, the new definitions are implementation-independent and allow a wide range of concurrency control techniques, including locking and optimism. Furthermore, our definitions handle predicates in a correct and flexible manner at all isolation levels. Thus, they meet the goals of the ANSI-SQL standard.

[1] As stated in [14], this does not imply that a PL-3 transaction observes a consistent state since lower level transactions may have modified the database inconsistently; if we want a PL-3 transaction to observe a consistent state, lower level transactions must update the database consistently even if they observe an inconsistent state.

The paper also specified the behavior of systems that allow mixing of levels: users are allowed to choose the level for each transaction they run, and the system guarantees that each transaction is provided with the constraints of its own level, even when some transactions are running at lower levels.

Our approach is applicable to other levels in addition to the ones discussed in the paper. We have developed implementation-independent specifications of commercial isolation levels such as Snapshot Isolation and Cursor Stability, and we have defined a new level called PL-2+; the details can be found in [1]. PL-2+ is the the weakest level that guarantees consistent reads and causal consistency; it is useful in client-server systems [3, 1] and broadcast environments [25].

All of our definitions are implementation independent. This makes them suitable for use as an industry standard, since they do not preclude clever but unconventional implementations that either exist today or may be developed in the future. Instead they provide implementors with the opportunity to choose the best performing concurrency control mechanism for their environment.

Acknowledgements

We would like to thank Chandra Boyapati, Miguel Castro, Andrew Myers, and other members of the Programming Methodology Group for their comments. We are grateful to Dimitris Liarokapis and Elizabeth O'Neil for carefully reading the paper and helping us improve the specifications. We would also like to thank Phil Bernstein, Jim Gray, and David Lomet, for their helpful comments.

References

[1] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, MIT, Cambridge, MA, Mar. 1999.

[2] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control using Loosely Synchronized Clocks. In *SIGMOD*, San Jose, CA, May 1995.

[3] A. Adya and B. Liskov. Lazy Consistency Using Loosely Synchronized Clocks. In *Proc. of ACM Principles of Dist. Computing*, Santa Barbara, CA, Aug. 1997.

[4] D. Agrawal, A. E. Abbadi, and A. K. Singh. Consistency and Orderability: Semantics-Based Correctness Criteria for Databases. *ACM TODS*, 18(3), Sept. 1993.

[5] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed Multi-version Optimistic Concurrency Control with Reduced Rollback. *Distributed Computing*, 2(1), 1987.

[6] *ANSI X3.135-1992, American National Standard for Information Systems – Database Language – SQL*, Nov 1992.

[7] B. R. Badrinath and K. Ramamritham. Semantics-Based Concurrency Control: Beyond Commutativity. *ACM TODS*, 17(1), Mar. 1992.

[8] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proc. of SIGMOD*, San Jose, CA, May 1995.

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

[10] P. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models using ACTA. *ACM TODS*, 19(3), Sept. 1994.

[11] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, Fifth edition, 1990.

[12] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proc. of SIGMOD*, Montreal, Canada, June 1996.

[13] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Database. In *Modeling in Data Base Management Systems*. Amsterdam: Elsevier North-Holland, 1976.

[14] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.

[15] R. Gruber. *Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases*. PhD thesis, M.I.T., Cambridge, MA, 1997.

[16] R. Gruber, F. Kaashoek, B. Liskov, and L. Shrira. Disconnected Operation in the Thor Object-Oriented Database System. In *IEEE Workshop on Mobile Comp. Systems*, 1994.

[17] M. P. Herlihy. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *ACM TODS*, 15(1), March 1990.

[18] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc of the ACM Symp. on Operating Sys. Principles*, Pacific Grove, CA., Oct. 1991.

[19] H. Korth, A. Silberschatz, and S. Sudarshan. *Database System Concepts*. McGraw Hill, 1997.

[20] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM TODS*, 6(2), June 1981.

[21] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. of SIGMOD*, Montreal, Canada, June 1996.

[22] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented dDBMS. In *Proc. of OOPSLA*, Sept 1986.

[23] P. O'Neil. The Escrow Transactional Method. *ACM TODS*, 11(4), Dec. 1986.

[24] Oracle Corporation. Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7, July 1995.

[25] J. Shanmugasundaram, A. Nithrakashyap, R. Sivasankaran, and K. Ramamritham. Efficient Concurrency Control for Broadcast Environments. In *SIGMOD*, Philadelphia, PA, June 1999.

[26] D. Terry et al. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. of SOSP*, Copper Mountain Resort, CO, Dec. 1995.