Concurrency Control Chapter Handbook of Database Technology

By Patrick O'Neil <poneil@cs.umb.edu>
University of Massachusetts at Boston

Introduction

The need for ***Concurrency Control***, a protocol to avoid interference in database updates by concurrently executing processes, was recognized in the early 1960s. The earliest database applications in which this need became evident were: (1) The Sabre System for tracking Airline Reservations [GRAY, Sabre], and (2) A product known as WYCOS 2 that was implemented on the GE Computer Integrated Data Store (IDS) for an application at Weyerhauser Corporation [BACH]. The first published papers to explain concurrency control didn't appear until more than ten years later, in the mid-1970s. In this Chapter we will provide an overview of some of the major developments in concurrency control, which continue to see innovations integrated in commercial products even to the present day.

In this Chapter we will assume some knowledge of SQL (see Chapter 10 of this Text), as well as a general grasp of computer processors and disks.

1. Early Lessons in Concurrency Control

Let us start with a few basic ideas to illustrate the need for Concurrency Control.

1.1. Interference Problems of Concurrent Access

To begin, assume there are two computer processes, P1 and P2, performing banking application requests by two tellers in a bank. In simplest terms, we can assume that the work performed will sometimes involve ***Reads*** of pieces of accounting data of the bank, as when a teller reads the balance of a customer account at the request of the customer, or ***Writes*** of pieces of accounting data of the bank, as when a teller writes a new home address for the holder of a bank account. The pieces of data we Read or Write are generically referred to as ***data items***, symbolized by the letters A, B, C, D, etc. The notation of a Read operation of the data item A performed by P1 is $R_1(A)$ and the notation for a Write operation on data item B performed by P2 is denoted by $R_2(B)$.

Now the sort of error that can occur in concurrent access is illustrated by the following example, a sequence of operations that is usually called a ***History***:

H1: $R_1(A)$ $R_2(A)$ $W_1(A)$ $W_2(A)$ LOST UPDATE HISTORY EXAMPLE

Note that the history H1 does not actually specify particular rows or the values read or written by the processes P1 and P2. These details are removed so that a simple mechanism that merely keeps track of data item identifiers and agents that Read and Write them can detect the types of concurrency errors dealt with here.

Example 1. An Interpretation of H1. To show how the History H1 can represent a possible error, known as ***Lost Update***, we provide what we refer to as an ***Interpretation*** of H1. We assume that A represents a bank balance, and that two co-holders of the account, acting in distinct processes P1 and P2, are concurrently depositing amounts of money to the account balance, say $30 and $40 res-

pectively, where the balance begins with value $100. We use a form of ***extended notation*** to display an interpreted history of H1, named H1', which provides values of data items Read or Written, to show the problem:

H1': $R_1(A,100)$ $R_2(A,100)$ $W_1(A,130)$ $W_2(A,140)$

As we see in H1', the final value of A is the value 140 ($140). But if the concurrent access had not taken place, that is if P1 had gone first and performed all its operations, and after P1 was done, P2 had performed its operations, the final value would have been $170. This is known as a ***Serial History***, S1(H1'):

S1(H1'): $R_1(A,100)$ $W_1(A,130)$ $R_2(A,130)$ $W_2(A,170)$

Similarly, if P2 executed all its operations first and then P1, we would have:

S2(H1'): $R_2(A,100)$ $W_2(A,140)$ $R_1(A,140)$ $W_1(A,170)$

Thus S1(H1') is a serial history where P1 goes first, and S2(H1') a serial history where P2 goes first, and both of them give a correct result. It seems clear that an error occurred in the concurrent execution of H1', and that this error was caused by ***Interference*** in concurrent access to the data: It occurs because P2 remembers the value of A as 100, even after P1 had already added 30 to make it 130, P2 then performs an update that wipes out P1's effect.

There is no way an unaided programmer can prevent the concurrency error of H1'. For example, one cannot avoid the problem by insisting P2 Read the data item A again immediately before Writing. In fact, this is already happening, since the last operation by P1 prior to $W_2(A,140)$ is $R_2(A,100)$. The problem is that process switching has taken place immediately after $R_2(A,100)$, allowing the interposed write by P1 to follow. Processes in a time-sharing system will commonly need to WAIT when they perform a Read requiring an I/O, so it's not possible for the application by itself to avoid an interposed Write $W_1(A,130)$. The only way to avoid this error by supplying a ***Concurrency Control*** method supported by the Database System itself.

The first concurrency control method implemented in practice was known as Two-Phase Locking. Typically, locks were taken on data items on behalf of the agent reading or writing them, giving the agent a form of privileged access to the data, and the locks were released when the agent had completed all its actions. The application performing the balance knows when it is done and can signal this by a ***Commit*** action, or in some cases, say if a withdrawal would bring an account balance below zero, an ***Abort*** action, which signals that all updates are to be reversed. It turns out to be inappropriate to think of a Process as the agent that does this, since the same process might be used multiple times to perform a deposit (or withdrawal) from different bank balances. Instead, early developers invented something called a ***Transaction*** (executing in a process, or in more modern systems in a more lightweight thread) to collect locks of various kinds until the application ends the Transaction's life with a Commit or Abort. The Process lives a long time and is likely to support a long sequence of different Transactions before the process ends.

With a new Commit operation for a transaction T_i with notation C_i and an Abort operation with notation A_i, the original H1 history would now be written as:

H1: $R_1(A)$ $R_2(A)$ $W_1(A)$ $W_2(A)$ C_1 C_2 Lost Update Anomaly

Of course we have already shown H1 has a concurrent execution error, and in what follows we will refer to concurrency errors of this kind as ***Anomalies***.

1.2 Serializability

Early researchers saw that a criterion was needed to determine when a history of operations could give rise to an Anomaly, and when it could be guaranteed not to do so. We have already encountered the touchstone that was used to determine a "correct" transactional history, namely a "Serial History". The criterion for a correct transactional history turned out to require the following steps.

1. Define conditions under which a history is serial;
2. Define conditions under which two histories are (meaningfully) equivalent;
3. Define a history to be serializable (correct) exactly when it is equivalent to a serial one.

A (Committed) ***Transaction*** $\mathbf{T_i}$ is defined as a sequence of operations, consisting of elements from the set $\{R_i(X), W_i(X), C_i\}$, with a temporal order $<_i$ and a Commit operation C_i in the terminal position. The idea is that an application program requests the transaction's operations in a given order, ending with a Commit, and this order cannot ever be modified in transforming a history consisting of multiple transactions. Of course there can also be an Aborted transaction T_i ending with an A_i.

A ***Complete History*** **H** (in a centralized database -- no parallelism) is defined to contain all the operations from a set of Committed Transactions $\{T_i, i = 1,n)$ and an ordering of those operations $<_H$ such that for i = 1,n, $<_i \subseteq <_H$, i.e., *the order of operations in original transactions T_i are maintained in the history H.*

The reason we deal with a complete history of committed transactions, with no partial transactions allowed, is this. The definition is intended to measure the success of a transactional scheduler imposing a concurrency control to maintain a "Serializable" history as operations are submitted to the scheduler in order as the concurrently executing transactions submit them. The scheduler can delay certain transactional operations, or as sometimes becomes necessary, Abort a transaction, to maintain correctness. At any time the history might contain certain transactions that hasn't yet committed and in fact cannot Commit if the history is to remain serializable. The solution used by the scheduler will be to Abort such transactions. Thus we do not want to consider uncommitted transactions as counting against the scheduler's effectiveness, and this is why we only deal with Complete Histories in defining a serializable history.

We have already seen examples of serial histories S1(H1') and S2(H1') in Example 1 of Section 1.1. Here is a more precise definition.

A complete history H is a ***Serial History*** if and only if, for every two transactions T_i and T_j in H, either all operations of T_i appear before all operations of T_j or vice-versa, all operations of T_j appear before all operations of T_i.

Of course this definition of a Serial History H implies that any transaction in H will have all its operations occurring in one contiguous block of time, with no interleaved operations of other transactions. Thus a Serial History always avoids any concurrent interference between transactions.

Now we want to say that any history H that is in some sense ***equivalent*** to a Serial History S(H), is Serializable as a result, and this is the only way in which H will be **correct** and free from concurrent execution Anomalies. A number of definitions of equivalence have been defined, but the one that the early researchers settled on is called ***Conflict Equivalence***.

An operation (Read or Write) by transaction T_i and a second operation by T_j in a history H are said to ***Conflict*** or to be ***Conflicting Operations*** exactly when they act on the same data item and at least one of them is a Write operation.

Here is justification for the definition of conflicting operations. We claim that two successive conflicting operations cannot be commuted to occur in the reverse order without the likelihood of changing their effect within the history. One way to define Conflict Equivalence between a history H and a serial history S(H) is the ability to find a series of commutes of adjoining operations in H to generate S(H). So why do we define Conflicting Operations as we do?

First, we note that we can never commute two operations of a single transaction, so that explains why conflicting operations must be from different transactions.

Second, the two operations access the same data item because there is clearly no effect if the pair $R_i(A)$ $W_j(B)$ (say) is reversed to become $W_j(B)$ $R_i(A)$.

Third, two Read operations on the same data item can always commute: $R_i(A)$ $R_j(B)$ and $R_j(B)$ $R_i(A)$ have the same effect;

Fourth, consider the two adjoining operations $R_i(A)$ $W_j(A)$ and the reverse pair $W_j(A)$ $R_i(A)$. In these two orders, $R_i(A)$ is likely to read different values (assuming that $W_j(A)$ changes the value of A from what it was earlier).

Finally, the adjoining operations $W_i(A)$ $W_j(A)$ and the reverse pair $W_j(A)$ $W_i(A)$ cause data item A to end up with different values, assuming the two Writes are not creating identical values.

Now it turns out that a ***Serializable History*** can contain pairs of conflicting operations as long as the history doesn't contain a cyclic sequence of conflicting pairs. We give two examples of such a cyclic sequence of conflicts below.

Let us see why the Lost Update Anomaly, repeated below, is not serializable.

H1: $R_1(A)$ $R_2(A)$ $W_1(A)$ $W_2(A)$ C_1 C_2 Lost Update Anomaly

All operations of T_1 and T_2 are on the same data item A, and there are three different conflicting pairs of operations in H1, but we will only consider two. The first conflicting pair is: (1) $R_1(A) <_{H1} W_2(A)$ (where $<_{H1}$ gives the time order from left to right), and the second conflicting pair is: (2) $R_2(A) <_{H1} W_1(A)$. Since both of these pairs must remain in the same order in any equivalent serial history S(H) (conflicting pairs cannot be commuted), then based on S(H) pair (1), all operations of T_1 must come before all operations of T_2. But because of pair (2), all operations of T_2 must come before all operations of T_1. There is a contradiction, and this shows that any assumption that H1 has an equivalent serial history must be false. H1 is thus not serializable, which we have already concluded in the interpretation of H1' in Section 1.1.

Here is another well-known history example that leads to a concurrency Anomaly:

H2: $R_1(A)$ $R_2(B)$ $W_2(B)$ $R_2(A)$ $W_2(A)$ $R_1(B)$ C_1 C_2 Inconsistent Analysis Anomaly

H2 contains the conflicting pairs: (1) $R_1(A) <_{H2} W_2(A)$ (the first and fifth operations) and (2) $W_2(B) <_{H2} R_1(B)$ (the third and last operations), and as above, this implies that in any S(H2) all operations in T1 must come before T2 and vice-versa, a conflict cycle that shows S(H) does not exist. We can show the same fact using an interpretation H2':

H2': $R_1(A,100)$ $R_2(B,100)$ $W_2(B,50)$ $R_2(A,100)$ $W_2(A,150)$ $R_1(B,50)$ C_1 C_2

In H2', T_1 is moving $50 from savings balance B to checking balance A, while T_2 is adding up the account balances A and B, and finding a total of $150. But in any serial schedule, if T_1 preceded T_2 it would sum $100 + $100 to get $200, and if T_1 succeeded T_2, T_1 would sum $150 + $50 to get $200. T_2 only sees a sum $150 in H2' because of concurrent interference; of course such error might be crucial for an account holder who is having bank balances checked to apply for a loan.

2. Two-Phase Locking (2PL)

The material we have presented so far doesn't provide us an efficient algorithm by which a database system scheduler can coerce a series of data item Read and Write requests to produce a serializable history. Perhaps we should first ask ourselves why the scheduler doesn't simply impose a fully Serial schedule on the operations submitted.

A scheduler can impose a serial schedule by accepting the first operation submitted by (say) transaction T_1 and then allowing only operations submitted by the T_1 to execute; operations by other transactions are simply queued up until T_1 Commits. Then the scheduler can begin to execute queued operations of the next transaction, say T_2, until that transaction also Commits, and so on. In this way the scheduler guarantees a serial schedule, which is correct by definition.

The reason this approach is not used in practice is that it is terribly inefficient. Most transactional applications run very short CPU bursts (fractions of a millisecond) between I/O requests from disk (which take multiple milliseconds). While the process running a transaction is in I/O wait, the OS must look for another process to execute in the CPU. If the only processes running in the database system are the ones executing transactions, then if the database scheduler imposes a serial schedule, there will be no other transaction to run when the single running transaction goes into an I/O wait. This can mean that only a few percent of the CPU will be utilized. If we allow concurrent execution instead, and disperse the data accessed by the transactions over a large number of independent disks, then the throughput in transactions per second will be greatly improved.

The algorithm used in all the early database schedulers to support concurrently running transactions while guaranteeing a serializable schedule, is known as Two-Phase Locking (2PL). The concepts of Serializability and Two-Phase Locking were both introduced in a 1976 paper by four IBM practitioners [EGLT]. All current textbooks that cover transactional theory cover both these concepts in detail.

In ***Two-Phase Locking***, locks are taken and released on data items by the scheduler according to the following rules.

(1) Before a transaction T_i can Read a data item A, $R_i(A)$, the scheduler tries to Read Lock the item on its behalf, $RL_i(A)$; similarly, before a Write of A, $W_i(A)$, the scheduler tries to Write Lock A, $WL_i(A)$.

(2) If a conflicting lock on the data item A already exists, the requesting Transaction must WAIT (a process Wait), until the required lock can be taken on its behalf by the scheduler.

(Conflicting locks correspond to conflicting operations, defined in Section 1.2: two locks on a data item conflict if they are attempted by different transactions and at least one of them is a WL).

(3) **Two-Phase Rule.** There are two phases to locking, the growing phase and the shrinking phase (when data items are Unlocked: $RU_i(A)$); The scheduler must ensure that a transaction can't enter the shrinking phase (by releasing a lock) and then start growing again (by taking a new lock).

The Two-Phase Rule implies that a transaction might release locks before Committing. But permitting applications to release locks early has always been a risky proposition, and in SQL all data item locks are released at one time, on Commit. We'll assume this behavior in what follows:

All locks of a transaction are released exactly when the transaction Commits.

Of course when all locks are released at the time of Commit, the Two-Phase Rule is still valid -- the shrinking phase occurs at Commit time, so no new locks are taken after the shrinking phase starts.

Note too that *a transaction will never have conflicts with its own locks!* If T_i has taken a lock $RL_i(A)$, then it can later get a lock $WL_i(A)$, so long as no other Transaction T_j holds a lock on A (of course that this would need to be a Read lock, $RL_j(A)$, or it could not be held while T_i held $RL_i(A)$). Note too that if T_i holds a lock $WL_i(A)$, it won't need to later take the lock $RL_i(A)$ to perform a Read, because a Write lock implies a Read lock.

If locking is to guarantee serializability for the resulting history, it must be able to guarantee that a cycle of conflicts will never occur. We see that the first transaction T_1 to lock a data item A forces any transaction T_2 that attempts to take a conflicting lock on A at a later time to WAIT until after T_1 Commits. But what if T_2 already holds a lock on data item B before it WAITs, and T_1 later needs to take a conflicting lock on B? This would mean a cycle of conflicts; but given the WAIT rule when attempting to take a conflicting lock, this will mean both transactions will WAIT for the other to release the lock they need, and since each transaction WAITs, neither transaction can ever leave WAIT state. This is a ***Deadlock***. The scheduler is designed to recognize when such a deadlock occurs, and force one of the Transactions involved (a ***Victim***) to Abort.

We give an example of such a deadlock based on the Inconsistent Analysis Anomaly history covered in Section 1.2.

H2: $R_1(A)$ $R_2(B)$ $W_2(B)$ $R_2(A)$ $W_2(A)$ $R_1(B)$ C_1 C_2 Inconsistent Analysis Anomaly

We rewrite history H2 as HL2, with appropriate locking operations, and explain the behavior the scheduler will take as operations of this history are submitted.

HL2: $RL_1(A)$ $R_1(A)$ $RL_2(B)$ $R_2(B)$ $WL_2(B)$ $W_2(B)$ $RL_2(A)$ $R_2(A)$ $WL_2(A)$ (This lock attempt fails, since $RL_2(A)$ conflicts with $WL_1(A)$; T_2 must WAIT for T_1 to Commit or Abort and release locks) $RL_1(B)$ (This lock attempt fails since it conflicts with $WL_2(B)$ so T_1 must WAIT for T_2 to complete; but this is a deadlock! We assume the scheduler will choose T_2 as victim) A_2 (now $RL_1(B)$ will succeed) $R_1(B)$ C_1 (As is common, we assume the application running T_2 sees the Deadlock Abort error message, and performs a RETRY of T_2, assigned a new Transaction number T_3 by the scheduler) $RL_3(B)$ $R_3(B)$ $WL_3(B)$ $W_3(B)$ $RL_3(A)$ $R_3(A)$ $WL_3(A)$ $W_3(A)$ C_3.

Note that there can be more than two transactions involved in a Deadlock when there is a cycle such as the following: T_i waits for T_j waits for T_k waits for T_i.

We define a ***Waits-For Graph***, ***WFG***, as a directed graph maintained by the scheduler to test for deadlocks, as follows. The Vertices of the Waits-For Graph correspond to active transactions, currently being executed, T_i, T_{i+1}, etc.; there is a Directed Edge of the WFG, T_i -> T_j, exactly when transaction T_i is waiting for a lock on some data item on which T_j holds a conflicting lock. The scheduler can test for a deadlock cycle of edges each time a new edge is added to the WFG.

Note that non-serializable histories don't necessarily cause deadlocks. In many cases, a ***delay*** in granting some transactional operation lock (and succeeding operations of that transaction until the original lock is granted) will result in a serializable history. Consider, for example, the history H3, a simple reordering of operations of H2, which is still inconsistent.

H3: $R_2(A)$ $W_2(A)$ $R_1(A)$ $R_1(B)$ $R_2(B)$ $W_2(B)$ C_1 C_2 Inconsistent Analysis Anomaly 2

A cycle of conflicts exists with the two pairs of operations: $W_2(A) <_{H3} R_1(A)$ and $R_1(B) <_{H3} W_2(B)$, so H3 is not serializable. Here is a history H3' with ***extended notation*** providing values of data items Read or Written, that is the basis of an interpretation showing the same thing:

H3': $R_2(A,100)$ $W_2(A,50)$ $R_1(A,50)$ $R_1(B,100)$ $R_2(B,100)$ $W_2(B,150)$ C_1 C_2

In H3'. T_1 is summing the two bank balances A and B to 150, while T_2 is moving $50 from A to B, and the sum of the balances starts as $100 + $100 and ends as $50 + $150, a sum of $200 in both cases. We note that the situation here is that in moving money from one balance to another, T_2 must update one data item at a time, meaning there is an inconsistent state for the two data items at the time that T_1 evaluates the sum in the middle. This sort of inconsistency is forbidden by a property of transactions known as ***Atomicity***, where all Read and Write operations in a history must succeed or fail as a unit -- Atomically -- and results must be seen Atomically by all other transactions.

Here is the Locking schedule for H3, HL3.

HL3: $RL_2(A)$ $R_2(A)$ $WL_2(A)$ $W_2(A)$ $RL_1(A)$ (This lock attempt fails, since $RL_1(A)$ conflicts with $WL_2(A)$; all operations of T_1 must WAIT for T_2 to Commit (or Abort) and release locks) $RL_2(B)$ $R_2(B)$ $WL_2(B)$ $W_2(B)$ C_2 (Now the $RL_1(A)$ request is successful, and T_1 can continue) $R_1(A)$ $RL_1(B)$ $R_1(B)$ C_1.

We see in HL3 that no deadlock occurred, and T_1 saw the sum $50 + $150 because of being delayed.

Two-Phase Locking Guarantees Serializability

Nearly all textbooks that deal with database transactions give a proof that 2PL guarantees serializability. The idea of the proof is that a cycle of conflicting transactions, T_1, T_2,..., T_n, $T_{n+1} = T_1$, cannot exist in a history H produced by a locking scheduler, because this would imply there is a data item D_i unlocked by each T_i and then locked by its successor transaction T_{i+1} (this is the data item causing the conflict), and thus we must have $T_1 <_H T_2 <_H T <_H ... <_H T_n <_H T_{n+1} = T_1$. But this must mean that T_1 locks a data item D_n after it unlocks a data item D_1, which disobeys the Two-Phase Rule. Thus H is not produced by a locking scheduler, a contradiction, which shows that no such cycle of conflicts can exist.

3. Additional Transactional Considerations

There are a number of transactional issues that we won't be able to consider in great depth in the current chapter, but nevertheless represent important concepts that the reader may find of interest to learn more about in other chapters or database texts. In what follows, we include short sections describing a number of such issues.

3.1 Transactional Database Performance

Traditionally, transactional performance was a crucial factor in choosing a database product to be used by large organizations. Several ***Database Benchmarks*** exist, standard tests of transactional performance including carefully crafted database structures and (one or more) transactional "programs" to match a common type of commercial application [TPC-A,TPC-B,TPC-C]. These benchmarks require complex tuning, and the final one in particular was often run by the database companies themselves to provide confidence that their products offered good transactional performance.

This concern with performance is explained by a few factors that defined database systems for many years, many of which seem less and less important with modern developments in hardware. Here is a short outline of these defining factors.

Memory Cost and Disk Page Buffering

Data was historically held on Disk, which was much larger than computer memory, and the memory itself was extremely expensive. In about 1981, IBM responded to competition from IBM compatible memory producers by reducing IBM memory price to $10,000 per MByte ($2^{20}$ = 1,048,576, roughly a million bytes); this was considered a monumental event at the time! What this meant was that anyone with a large database, containing perhaps a GByte (2^{30} or a billion bytes) of data, would store the data on disk and bring it into memory in small blocks, usually disk pages of about 4 KBytes (2^{10} or a thousand bytes), because only in memory could the data be properly interpreted and updated. The disk pages were brought into what were known as ***Memory Buffers***, each buffer containing one 4 KByte page of data. Typically, the entire set of memory buffers took up perhaps 10 MBytes of memory, only 1/100 of the total GByte of data on disk; this meant that disk data had to be read into memory buffers in small portions, and unchanged data was usually (not always) later dropped from buffer when it was no longer needed, to make space for the next page read from disk. Data pages that had been updated while in buffer, would first need to be written out to replace the prior version of the page on disk, and then (usually) dropped from buffer.

It was recognized early on, however, that disk pages that were very popular, meaning that they were frequently referenced, would more profitably remain in memory buffer. This was a simple trade-off between memory space to hold the disk page (expensive) and the use of a disk arm to access the disk page on disk at the next reference (also expensive, especially for a popular page) [GRAYPUT]. The algorithm that determined which data pages were dropped from buffer and which remained depended on the frequency of access to the data page, and was known as the ***Page Replacement Policy***. Several such policies were studied in the early 1960s, and the one that had the best characteristics was called the ***Least Recently Used (LRU)*** page replacement policy, which dropped from buffer the page that had not been accessed for the longest period. Some other page replacement policies have been developed since that time [LRU-K], but most database system products still use the LRU algorithm.

One important observation is that improvements in the disk buffering algorithm are becoming less meaningful as memory becomes cheaper and typical processor memories pass the 2 GByte barrier. (Up until the 1990's, even the most expensive processors didn't have the ability to address as much as 2 GBytes, since a few bits of the 32-bit address word had been used for other purposes.) Large memories

makes it possible to support what is known as a ***Memory-Resident Database*** for many transactional applications, a concept that was developed theoretically during the 1980s [DEWITT]. But memory-resident databases are not as popular as one might expect for a number of reasons. First, transactional databases with data on disk are a well-solved problem, and today's hardware ensures there is little cost to using disk-resident data. In fact some major on-line commercial companies are simply using a large number of parallel processors and storing data on a file system with each new product added to an order, passing up the better performance of transactional processing. This is not an approach that a normal company should attempt, however; on-line orders required so much development effort when they first emerged that the developers thought they might as well create a custom approach to saving the data and recovering from machine and disk crashes, a difficult problem that they had to solve in any event.

What certainly is true for nearly all companies, however, is that the cost of processing transactions, even in large banks, is dropping down to a small fraction of the cost of providing tellers with terminals to access the system. This became clear some years ago when certain database benchmarks that wanted to cost all aspects of transaction processing, had to reduce the number of terminals and the thinking time between transactions on a single terminal to an unrealistically small average of ten seconds. With the prior 100-second think time, the cost of even the cheapest terminals would have greatly exceeded the cost of the transactional system.

Indexing

While speaking of database performance, we must mention that any column by which a row is commonly accessed (as in SELECT BALANCE FROM ACCOUNTS WHERE ACCTID = 'A273196') must be ***indexed*** for quick retrieval. Any transaction that retrieves rows based on values of a single column that isn't indexed will need to search the entire table to find the rows selected. An index typically orders the Keyvalues on which rows are to be looked up (ACCTID Keyvalues for a unique index on ACCOUNTS or MAKE_OF_CAR Keyvalues for an index on AUTOS in a Registration table). The most common index is known as a ***B-tree index*** (more properly, a B^+-tree index, but the simpler term is more commonly used), and can be thought of as allied to binary lookup tree on Keyvalues in memory; however B-tree nodes lie on disk and contain many entries, so the ***Fanout*** of the B-tree is quite large compared to a fanout of two for binary indexes. As rows are inserted, updated, or deleted from a table, all index entry changes associated with these operations are immediately made. For example, when a new row is inserted in a table, entries for the row will be inserted in all associated indexes.

It turns out in what follows that index entries need to be locked to guarantee serializability. We will deal with this in Section 4, when we discuss the Phantom Anomaly.

3.2 Transactional ACID Properties

The ***ACID Properties*** for database transactions were originated by Jim Gray beginning in [GRAY81]. The ACID properties can be thought of as a set of guarantees offered to application programmers who use transactions in data access rather than an unprotected form of data access such as calls to an OS File System. There are four properties given, from which the ACID acronym is derived: ***Atomicity***, ***Consistency***, ***Isolation***, and ***Durability***.

Atomicity. This is a guarantee that the set of record updates that make up a transaction are indivisible, that either all occur or none occur, and no agent can observe an inconsistent result (including other transactions running concurrently and users waiting for messages output by the program). If an access fails,

or any other error occurs, the entire transaction must Abort, and no effect of the transaction will remain. If the program encounters a conceptual error (such as some bank balance being too low to make a required payment), it has the ability to bring about an Abort by itself (the programmer can arrange this with a SQL statement ROLLBACK WORK). Without the ROLLBACK WORK statement, a program that found it had to back out its changes would need to reverse complicated updates by programmatic logic, which would be extremely error-prone.

Consistency. We assume that a database and transactions acting on it must obey a number of ***Constraints***[1] that define a legal state of the database. One such constraint might be that no bank balance can fall below zero; another might be that in a transfer between account balances, money is neither created nor destroyed. Many such constraints must be guaranteed by the transaction program acting in isolation rather than by the database system returning an error; the Consistency guarantee requires that concurrent transactions finding a legal (Consistent) database state, will leave it that way on Commit.

Isolation. This property guarantees that executions of concurrent transactions act as if they are isolated, that is, not interleaved in time. It is another way of saying that the history of transactions acts in a serializable manner. .

Durability. This property guarantees that once the program has been notified of the success of a transaction Commit, the transaction is guaranteed that the resulting changes will survive system failure. Thus the user may be notified of success (e.g., that a bank transfer has been successful, or that a withdrawal amount can be disbursed by the teller to the customer). This guarantee requires a ***Transactional Recovery*** capability to insure, after system failure, that all committed transaction results will be corrected on disk, even if updated pages were not all written out from buffers when transactions committed. We will discuss Transactional recovery in the next subsection.

Comment: Consistency is actually implied by the Isolation guarantee and the requirement that the transactions devised by the programmer perform legal changes of state when run in isolation.

3.3 Transactional Recovery

Transactional Recovery, or simply ***Recovery***, was mentioned in the previous Section 3.1 as supporting the ACID Guarantee of Durability. The motivation for Recovery is this. Recall that when pages are read from disk, they are stored in memory buffers, because it is only there that they can be updated. Unfortunately, memory is ***Volatile***, meaning that in a system failure, all memory contents will be lost. Now if a transaction transfers money from one bank account to another, and these accounts sit on two different disk pages that must be read into buffer and updated, then after one of the disk pages is written back out to replace its older version on disk, the system may fail before the second disk page is written out. If the disk contents of the database remain in this state when the database is restarted, it would mean an unbalanced transfer had occurred, with one account out of synch with the other: thus money would be either created or destroyed.

To avoid problems such as this, the database system writes Notes to itself, called ***Log Entries*** (or simply ***Logs***) that describe what updates are applied, and by which transactions, to rows on disk pages in memory (e.g., $W_i(A)$). The

---

[1] Many definitions use the term *Consistency Constraints* here, but Consistency Constraints in relational databases are constraints that are guaranteed by the database itself; a transaction breaking such a constraint would receive an error return, and the correct action would be to abort. No such built-in constraint guarantees that a transfer does not create or destroy money, for example.

sequence of Log Entries for transactional operations, as they occur, are placed together in a large ***Log Buffer*** in memory, and the Log Buffer is written out to the ***Disk Log*** from time to time; in fact the Log Buffer is written out to TWO disks at once, to guarantee stability even if one disk fails. In particular when transactions request a Commit, a Log entry noting the Commit (C_i) must be placed in the Log Buffer and written out to the Disk Log before the Commit acknowledgment can be returned to the calling program. Later, if the system fails in the middle of processing, when System ***Restart*** takes place the database system will read all the relevant Logs from disk and ensure that data updates that might not have reached disk will replace old versions of the disk data.

Recall that some disk pages are accessed so frequently that they are never dropped from memory buffer. The same consideration applies to updates of some disk pages. For example, if a row on the disk page contained a warehouse record providing quantity on hand for twelve ounce cans of a soft drink at the soft drink factory, and orders for that item were received every few seconds during the business day, it would be inefficient to write the row back to disk each time it was updated. Clearly the Log is made for situations like this, since it ensures that data that is not written out to disk will eventually get there if the system fails at some point during the day.

If we are writing the Disk Log at a high rate and it takes nearly as much time to recover from the log as it does to write it, we need to ensure that we don't need to recover through several hours of output because popular rows were never updated to disk during this entire period. To ensure that recovery time doesn't grow too long, Database systems provide an operation called ***Checkpointing***, which guarantees that popular pages that otherwise wouldn't be updated to disk for an extended period, are written out within some reasonable period set by the database administrator, say five or ten minutes. Then any part of the Disk Log more than ten minutes old can be garbage-collected, and recovery can start from a more recent position in the Disk Log.

3.4 Optimistic Concurrency Control

Although we explained at the end of Section 2 that Two-Phase Locking implies serializability, they are not identical concepts. It is easy to provide a history that is serializable but does not obey the 2PL protocol. E.g.:

H4: $R_1(A)\ W_1(A)\ R_2(A)\ W_2(A)\ R_1(B)\ W_1(B)\ C_1\ C_2$

H4 is serializable because it contains only two pairs of conflicting operations, $R_1(A) <_{H4} W_2(A)$ and $W_1(A) <_{H4} R_2(A)$, which do not create a conflict cycle; they both agree in implying that the conflict equivalent serial history will have all operations of T_1 precede all operations of T_2. However H4 is clearly not following the 2PL protocol, which would delay the operations of T_2, $R_2(A)\ W_2(A)$, until after all operations of T_1 were complete.

One alternative to the 2PL protocol is to have the scheduler immediately schedule each operation it receives, but remain aware of the conflicts and check frequently that no cycle of conflicts has arisen. If such a cycle has occurred, the scheduler must find a victim transaction to Abort. The process of deciding that a schedule (history up to this point) has no cycle, so that a transaction Commit can take place, is called Certification, and such a scheduler is called a ***Certifier*** [BHG]. Certifiers are often referred to as ***Optimistic Schedulers*** because they aggressively schedule operations, hoping that no non-serializable Anomalies will occur; Two-Phase Locking, on the other hand, is *pessimistic*, delaying all operations that conflict with a prior operation of a another transaction that is still active.

One reason that Optimistic schedulers were popular in research papers is that a performance study in 1985 [TAYGS] showed that transactions throughput in locking schedulers was limited by blocking due to conflicts, while Aborts were quite rare. Blocking can cause loss of performance when so many transactions are blocked (waiting for data items locked by another transaction) that the scheduler finds difficulty locating transactions to run during I/O Waits, and the percentage of CPU use is reduced. In the worst case, it is possible that, out of N concurrent transactions, only 1 is not waiting for a lock.

One problem with Certifier Schedulers, however, is that when one transaction optimistically reads the output of another (as in H4,...$W_1(A)$ $R_2(A)$...), if the first transaction Aborts at a later time, the second transaction that read from it must Abort as well, for it has depended on a value of the data item that no longer exists. This effect is known as Cascading Aborts, and does not occur in a 2PL scheduler.

No commercial database system has ever used an Optimistic scheduler of the kind indicated above, and a number of studies have indicated that performance of such schedulers are inferior to 2PL schedulers in resource-limited environments [AGRA]. In Section 6, however, we will cover a form of concurrency known as Snapshot Isolation that has some of the aspects of Optimistic concurrency, but without many of the drawbacks mentioned above.

3.5 Index Locking

As we previously indicated, database indexing uses a B-tree consisting of disk page nodes with a large fanout. All searches to the leaf level of the index start at a unique Root Node of the B-tree, and if a new index value is added for a newly inserted row, it may turn out that a large number of nodes of the tree will need to split to provide space for pointers to new nodes that result from node splits below. These splits can continue right up to the level of the root, and the 2PL protocol would therefore seem to require that a transaction making an insert to the tree needs to Write Lock the root node and hold the lock until Commit. But this approach would tend to make the index nearly useless for concurrent transactions that need to look up other rows with searches starting at the root. This problem has been addressed in numerous papers, however, and all commercial database systems provide an efficient alternative to two-phase locking in proceeding down a B-tree.

3.6 Distributed Transactional Processing

Up to now, we have been dealing exclusively with ***Centralized Transactional Processing***, where all work is performed on a single computer; the computer might have multiple CPUs with local cache memory, but it has shared memory where locking and cycle testing can be performed, and for our purposes we can assume that the number of CPUs on a single computer is immaterial.

An example of ***Distributed Transactional Processing*** is a very large bank, with perhaps a thousand local branches, each branch having its own computer where banking transactions are performed for local customers. The branch computers can be thought of as nodes in the network that communicate with one another, but the distributed database must be transparent in the sense that users should be able to interact with it as if it were one logical system. Customers from foreign branches might drop in at any time and draw on their account at their home branch. Indeed some customers might perform complex transactions that span multiple accounts at different branches. Distributed transactions of this kind provide a number of challenges that are not faced by centralized database systems.

For one thing, cycle testing in the Waits-For Graph can become orders of magnitude more difficult, since a cycle can extend over multiple independent nodes of the network. There are a number of theoretical solutions to this problem (known by names such as "Path Pushing"), but the complexity can become too much for any normal algorithm under exceptional circumstances. For this reason, database systems that support multiple node transactions have typically availed themselves of a ***Timeout*** solution. In this approach, requests to Read or Write data items are made to foreign nodes, and an ACKNOWLEDGE (ACK) message is awaited by the caller. If the ACK message takes longer than a certain time period (e.g.: 10 seconds), then it is assumed that a deadlock might have occurred and the transaction is Aborted and started over. Of course it might also happen that links or nodes of the network fail, but these problems are recognized by other means.

Another problem that arises in distributed transactions is how to perform a Commit. In a centralized computer, the system performs a Commit and awaits a response from the disk that the write of the disk log was successful. In the event of system failure before the transaction acknowledges and notifies the user, the result is uncertain, but we can rest assured that the transaction ended up in a well-defined state: either the Commit Log Entry reached the Disk Log, in which case the Transaction has succeeded in its Commit and this will be reflected after Restart, or the Commit Log Entry did NOT reach the Disk Log and the Transaction failed, so the result on Restart will be an Abort, wiping out all traces of the Transaction that might have reached disk.

Two-Phase Commit

Things are not so simple when a distributed transaction is being committed, because there is more than one node that can fail, and the two nodes need to agree on success or failure to Commit. Let us consider a case where a transfer of money is being made from one bank branch to another in a Transaction T, that consists of a transaction fragment T_1 on the node that initiated the transfer and another transaction fragment T_2 that performs the corresponding action on the second node. Typically, the node that started the transaction will become the Coordinator of the distributed transaction T to guarantee the Commit. The worst thing that can happen is that one of the two fragments of the distributed transaction, say T_1, succeeds, while the other fragment, T_2, Aborts. Once a transaction Commits, all details of the transactional updates are typically lost, so there is no easy way to back out an unbalanced Commit.

To avoid this possibility, the Coordinator will initiate a Prepare request to all fragments on the distributed transaction. A ***Prepare*** action is comparable to a Commit action, in that it results in a Log Entry on disk that will guarantee a result should there be a system failure at the node where the Prepare action was taken. The difference is, that instead of creating a Log that guarantees on Restart that a Commit will be performed, Restart processing on encountering a Prepare Log will return the transaction to the state where it started, able to either Commit or Abort. The Coordinator will start by performing a Prepare on its own fragment of the transaction, T_1, and then it will send Prepare messages to all other nodes involved in the transaction, in this case the one supporting transaction T_2. If all these transaction fragment nodes ACK the Prepare message of the Coordinator, we will be said to have completed the First Phase of the Two-Phase Commit. If one of the transaction fragment nodes fails to ACK however, the Coordinator will assume that transaction fragment has failed (quite likely the node's system has failed), and the Coordinator will Abort its fragment and send messages to all other nodes involved, including the one that failed to ACK, to Abort their fragment as well. The Coordinator will also log an Abort message for this Transaction ID in its Durable Message Queue, so that if some node later recovers from a failure and brings its fragment transaction back to a Prepared

state, it will be able to ask the Coordinator node for the message it missed, learn that it was an Abort, and act accordingly.

If, on the other hand, all the other transaction fragment nodes ACK the Prepare request, the Coordinator will now send Commit messages to all of them, and log the Commit message in its Durable Message Queue. In this way, if some node has a System failure and later recovers its Prepared fragment, it will know enough to Commit when it requests this message from the Coordinator.

4. The Phantom Anomaly

A ***Phantom Anomaly*** can be defined as a history of transactional operations that leads to obviously non-serializable behavior, despite the fact Data Item Read Locks and Write Locks obey 2PL to guarantee serializability. Here is an example.

First Phantom Anomaly Example. There has never been a good notation for Phantom histories developed. Here is an example of a Phantom in an experimental notation, and an interpretation that clarifies it. Below, we deal with two tables associated with records of a bank, with associated columns listed:

Table Name	**Column**	**Column**
ACCT (Accounts in bank)	BAL (Account balance)	BRANCH (Branch of ACCT)
BRACCT (Branch table)	BAL (Branch balance)	BRANCH (BRACCT Branch)

The ACCT table holds bank accounts in multiple branches of a large bank: ACCT.BAL is an account holder's balance and ACCT.BRANCH is the branch name. The BRACCT table holds one row for each branch of the bank, and BRACCT.BAL represents the total of all ACCT balances for the branch named in BRACCT.BRANCH.

In the notation that follows, P1 and P2 are two predicates: P1 restricts ACCT rows to ACCT.BRANCH = 'SFBay' (think of the SQL clauses: FROM ACCT WHERE ACCT.BRANCH = 'SFBay'), and P2 restricts BRACCT.BRANCH to 'SFBay' (SQL: FROM BRACCT WHERE BRACCT.BRANCH = 'SFBay'). Experimental notation in H5 below: PR is a predicate read operation, and I is an insert operation.

H5: PR_1(P1, SUM(ACCT.BAL)) I_2(A IN P1) R_2(BRACCT.BAL IN P2) W_2(BRACCT.BAL+A.BAL) C_2 R_1(BRACCT.BAL IN P2) {ERROR: WRONG TOTAL} C_2

INTERPRETATION: T_1, having summed the balances of all accounts from the SFBay branch of a bank ("PR_1(P1, SUM(ACCT.BAL))" is equivalent to the SQL statement: SELECT SUM(ACCT.BAL) FROM ACCT WHERE ACCT.BRANCH = 'SFBay'), needs to WAIT for I/O in order to read the total branch balance from the same branch. While T_1 WAITS, T_2 inserts a new account A into the SFBay branch, then adds A.BAL (say with value = 100) to BRACCT.BAL, updating the row and committing. As a result, T_1 will see the wrong BRACCT.BAL value when it performs its access. This could not have happened in any serial schedule, so the history is non-serializable, but all data item locks were scrupulously taken and failed to prevent the Anomaly.

The Anomaly occurred because when T_1 locked all the rows in ACCT with BRANCH = 'SFBay' while taking the SUM of ACCT.BAL, those locks did not prevent the insert of a new row of this type by a different transaction T_2.

The existence of phantoms was reported in [EGLT76], although the original Phantom Anomaly considered was somewhat different.

A Second Phantom Anomaly Example. Assume T_1 deletes one of the rows in the ACCT table for BRANCH = 'Berkeley'. The way that DB2 handled row deletes was to

perform the delete completely (including all index entries for the row), under the assumption that T_1 would most likely Commit, so the system wouldn't have to come back later to access and correct the disk pages involved unless T_1 ended with an Abort. (For the same reason, DB2 didn't leave a "Deleted Stub" record behind: it simply made the row disappear in the table.) Now if another transaction, T_2 is given the job of summing all the balances in the ACCT table for the same BRANCH while T_1 is still active, T_2 must somehow know enough to WAIT when it tries to look through all the Index entry for BRANCH = 'Berkeley'; T_2 cannot proceed with the assumption that the missing row entry has been deleted, since T_1 is still active and may Abort. But how will T_2 know enough to WAIT to access the index, since the row has been deleted and all index entries for the row have been deleted as well. There seems to be nothing for data item locking to act on. Thus, this was called a Phantom.

An early method of Phantom prevention was defined called ***Predicate locking***. The following paraphrase of a definitions from [BBGMOO] mirrors the discussion of phantoms and predicate locks in [EGLT76] and in Section 7.7 of [GR97].

> A *Read or Write predicate lock* is taken on the *potential set of rows* retrieved under the WHERE predicate of any SQL statement. The predicate lock covers all rows that might ever satisfy the WHERE predicate, including those already in the database and any that an Insert, Update, or Delete statement would cause to satisfy the predicate. Two predicate locks by different transactions conflict if at least one is an W lock (with a potential set of rows determined by a WHERE clause of an SQL update statement[2]), and the two predicates might be mutually satisfied by a common row, even when that row does not currently exist in the database.

Two problems caused predicate locking to be abandoned by System R [CHETAL81]:

> (1) determining whether two predicates are mutually satisfiable is difficult and time-consuming; (2) two predicates may appear to conflict when, in fact, the semantics of the data prevent any conflict, as in "PRODUCT = AIRCRAFT" and "MANUFACTURER = ACME STATIONARY COMPANY". (The presumption here is that a stationary company doesn't produce any aircraft.)

Indeed, it seems that predicate locking cannot avoid being pessimistic, seeing potential conflicts where none exist, because the R and W predicate locks are both taken in advance, on *potential sets* of rows, without any data-based knowledge of what conflicts are actually possible. In fact two Predicates that restrict different columns on the same Table will always be assumed to conflict. Thus if a Write Predicate is taken to update the balance of an ACCT row with ACCTID = 'A11234791' while an account holder at the bank asks for the sum of balances of the three accounts in his name, with a Read Predicate is taken on ACCT where NAME = 'John Smith', the two transactions will be assumed to conflict. A conflict in this case is unlikely, and when one considers the thousands of transactions per hour in an average bank that will be made to WAIT because of inappropriately assumed conflicts, it becomes clear why Predicate locking was abandoned.

The solution to the problem of how to efficiently prevent Phantoms was first published in papers by an IBM researcher [MOHAN90, MOHAN92], which provided techniques known as Key-Value Locking (KVL Locking) and Index Management Locking (IM Locking).

---

[2] This is a generic update statement, which may be an Update or Delete in this case. The definition is vague as to how an Insert statement with no meaningful WHERE clause can conflict with a Select statement WHERE predicate, but we should accept that this is intended.

The assumption underlying both KVL Locking and IM locking was that the great proportion of Queries and Updates involve the use of indexes. When no index is involved in one of these operations, this means that all rows of a table must be examined by a tablescan in order to determine what rows to update or retrieve. In that case a lock on the entire Table while the table is being scanned becomes reasonable. When indexes are being used for lookup, however, KVL or IM locks on index entries are taken to prevent Phantoms. We will concentrate on KVL locking in our explanation.

KVL Locking for First Phantom Anomaly. The full protocol is quite complex, so we will simply explain how to prevent the First Phantom Anomaly Example given at the beginning of this Section, using a simplified KVL protocol. Here is the history given in that Example.

H5: PR_1(P1, SUM(ACCT.BAL)) I_2(A IN P1) R_2(BRACCT.BAL IN P2) W_2(BRACCT.BAL+A.BAL) C_2 R_1(BRACCT.BAL IN P2) {ERROR: WRONG TOTAL} C_2

We assume there is an index on BRANCH in the ACCT table, which has multiple row pointers for each keyvalue (i.e., multiple ACCT rows for BRANCH = 'SFBay').

The first operation in H5, PR_1(P1, SUM(ACCT.BAL)), will take a KVL Read lock on the Keyvalue for BRANCH = 'SFBay' in this index (as well as Read locks on all rows accessed through this index). The second operation, I_2(A IN P1), will attempt in this simplified KVL protocol to take a Write lock on the Keyvalue for BRANCH = 'SFBay'[3], but will be unable to proceed because of a conflict with the earlier Read lock held by T_1. Thus T_1 will continue uninterrupted, finding the matching total balance in BRACCT.BAL for BRANCH = 'SFBay'. Only after this is finished will the I_2(A IN P1) operation be accomplished. In this way, the Anomaly of the Example will be prevented.

This simple example doesn't do justice to the flexible KVL locking protocol. One of the advantages of KVL and IM locking is that the pessimistic approach found in Predicate Locking is avoided. Each row updated with a KVL Write lock is treated separately in seeking conflicts, so there is no assumption that conflicting operations on predicates of different columns will necessarily conflict. A conflict will be found if and only if one of the rows being updated would specifically interfere with one of the rows being scanned by a conflicting read or being scanned by a conflicting update. Thus there would be no conflict found between an update to the price of "PRODUCT = AIRCRAFT" and scan of the prices of "MANUFACTURER = ACME STATIONARY COMPANY".

KVL Locking for Second Phantom Anomaly Example. We assumed T_1 deletes a single row in the ACCT table for BRANCH = 'Berkeley' (making the row disappear in the table). As we explained in the prior KVL Locking Example, an update such as this will take a Write lock in the BRANCH index on the Keyvalue for BRANCH = 'Berkeley'. Following this, when another transaction, T_2 is given the job of summing all the balances in the ACCT table for the same BRANCH while T_1 is still active, T_2 will find it must WAIT to take a Read lock on the Keyvalue for BRANCH = 'Berkeley' until T_1 completes its work, either by Commit or Abort. Thus the Phantom problem is solved.

---

[3] In the non-simplified protocol an Insert such as this will take a somewhat weaker lock, an IW (Intention to Write) lock, on BRANCH = 'SFBay'; this lock still makes a Read Lock WAIT, but two IW locks can be held at once, so it will be possible to Insert more than one row in Branch = 'SFBay' concurrently.

All commercial database system products that currently guard against Phantom updates to the full extent use a variant of KVL/IM locking, although they might not call it by that name. To simplify the nomenclature in what follows, we will refer to this kind of locking as ***Phantom locking***, more specifically the taking of ***Phantom Write locks*** and ***Phantom Read locks***. In this way we avoid using the Predicate locking terminology for the approach that was so disappointing in System R.

5. Isolation Levels (ILs)

The idea of Isolation Levels (ILs), first defined in ANSI SQL-92 and carried forward to ANSI SQL-1999 and -2003 [ANSI] , is that users might want to use a concurrency control method that will provide more concurrency, even at the expense of imperfect isolation. The highest (most restrictive) Isolation level, named SERIALIZABILITY, provides true serializable isolation, but lower levels (READ COMMITTED AND REPEATABLE READ) permit transaction executions to give anomalous results under some circumstances. It might seem strange to advocate a concurrency control approach that permits errors, but there are ways to restrict database application programs so that these errors do not occur (this fact is often poorly explained, however). The paper [TAYGS], mentioned earlier in the Optimistic Concurrency Control Section 3.3, showed there is serious loss of throughput in locking schedulers because of transactions being *blocked* and having to wait. Isolation Levels are one way to address this problem.

The concept of ANSI SQL Isolation Levels was based on a paper by IBM researchers published in 1976 [GLPT], where the Isolation Levels were called Degrees of Consistency. The idea was to be less strict about locking and thus allow more transactions to run. Degrees of Isolation were defined in terms of Locking behavior, with definition of phenomena that described the behavior, but ANSI SQL Isolation Levels were defined in terms of phenomena, and this caused a problem.

ANSI SQL-92/-99/-2003 Phenomena Definitions of Isolation Levels (ILs)

ANSI SQL Isolation Levels [ANSI] are defined in terms of described "Phenomena" P1, P2, and P3, which are English-language definitions of operation sequences that it was claimed could lead to anomalous behavior.

P1 (Dirty Read). Transaction T1 modifies a data item. Another transaction T2 then reads that data item before T1 performs a COMMIT or ROLLBACK. If T1 then performs a ROLLBACK, T2 has read a data item that was never committed and so never really existed.

P2 (Non-Repeatable Read): Transaction T1 reads a data item. Another transaction T2 then modifies or deletes that data item and Commits. If T1 then attempts to reread the data item, it receives a modified value or discovers that the data item has been deleted.

P3 (Phantom): Transaction T1 reads a set of data items satisfying some <search condition>. Transaction T2 then creates data items that satisfy T1's <search condition> and Commits. If T1 then repeats its read with the same <search condition>, it gets a set of data items different from the first read.

ANSI Isolation Levels were then defined in terms of whether the various Phenomena could occur in histories acting under these Levels, as shown in Table 1.

Table 1. ANSI SQL Isolation Levels Defined in terms of the Phenomena

Isolation Level	P1 Dirty Read	P2 Fuzzy Read	P3 Phantom
ANSI READ UNCOMMITTED	Possible	Possible	Possible

ANSI READ COMMITTED	Not Possible	Possible	Possible
ANSI REPEATABLE READ	Not Possible	Not Possible	Possible
ANSI SERIALIZABLE	Not Possible	Not Possible	Not Possible

ANSI explained that this definition was chosen because a generic statement of Isolation Levels was desired that didn't apply only to Locking. But in a 1995 paper [BBGMOO], the statements of these English-language Phenomena were shown to be flawed. In addition, a new Isolation Level known as Snapshot Isolation was exhibited, which seemed to be ANSI SERIALIZABLE by this definition but could be shown not to be truly serializable.[4]

Returning to the definitions of Degrees of Consistency [GLPT], and applying these definitions to the ANSI SQL Isolation Levels, the [BBGMOO] paper produced a locking definition of Isolation Levels that was not flawed. The idea behind this definition is to weaken how locks are held in concurrency efforts. Locks aren't always taken, and even when they are, many locks are released before EOT and new locks are taken after these locks are released. Therefore the Lower Isolation Levels do not provide Two-Phase Locking!

We define a ***short-term lock*** to be a lock that is taken prior to the Read or Write operation (R or W) and released immediately afterward. This is the only alternative to ***long-term locks***, which are held until EOT.

Then ANSI SQL-92 Isolation levels are defined as follows:

Note: Locks not taken Long Term are Short Term	Write locks on rows and Predicate Write locks are long term	Read Locks on rows, not Predicate Read locks are long term	Read locks on rows and Predicate Read locks long term
Read Uncommitted (Dirty Reads)	NA (Read Only)	No Read Locks taken at all	No Read Locks taken at all
Read Committed	Yes	No	No
Repeatable Read	Yes	Yes	No
Serializable	Yes	Yes	Yes

Note that Phantom Write locks are taken and held long-term in all isolation levels listed. The implications are explained below.

The following ANSI SQL statements can be given prior to a Transaction start to set the isolation level and specify whether the Transaction will perform updates:

set isolation level {**read uncommitted** | **read committed** | **repeatable read** | **serializable**}

set transaction {**read only** | **read write**}

It is intended that transactions of different users using different isolation levels can concurrently access the same data.

Read Uncommitted (RU). At this Isolation Level no locks of any kind are requested. Thus transactions running in Read Committed can read uncommitted data on which Write locks are held (nothing will stop the reader if it doesn't have to WAIT for a Read Lock). Of course the Transaction can thus access unbalanced data, but RU is just used to get a statistical idea of the data, say when the President

[4] The ANSI Definition of SERIALIZABLE was strengthened by a statement that it must provide concurrency that is generally recognized as serializability; so this ANSI Isolation Level is valid on that basis. All known citations of the ANSI ILs used only the Phenomena definition, however.

of a banking firm wants to know a ballpark figure of accounts on deposit. While a transaction running in Read Committed can return a message to the user, it is unable to Write any Data (when it is set to run in RU, it is automatically set to be a Read Only transaction); this reduces the possibility that an error will creep into the database itself.

Read Committed (RC). When a transaction running in Read Committed performs a Write, it will take long-term Write locks on both rows and predicates (holding them until EOT), but when the transaction performs a Read, it will take short-term Read Locks on rows and predicates (which are released immediately after performing the covered operation).

An Anomaly that can arise in a transaction running in Read Committed is a serious one: Lost Update (Example 1, Section 1.1):

H1': $R_1(A,100)\ R_2(A,100)\ W_1(A,130)\ C_1\ W_2(A,140)\ C_2$

Since Read Locks are released immediately, nothing stops the later $W_1(A,130)$; the later $W_2(A,140)$ would be stopped by the $WL_1(A,130)$, but since this is the last operation by this transaction, C_1 dropped this lock, so the value of A will be overwritten by T_2 for a value of 140, instead of both increments adding for 170.

SQL provides an alternative way to perform this logic, however. Instead of:

```
select A.balance into :val;   -- variable val to hold initial value of A = 100
val = val + incr;             -- incr variable of 30 will make val = 130
update A set balance = :val;  -- set A to 130
```

The program can perform the update in an indivisible manner:

```
update A set balance = balance + :incr;
```

Classical History notation has no equivalent to this, but we can postulate a new operation RW that reads a value from a data item and sets it to a newly incremented value.

H1*: $RW_1(A,100,130)\ C_1\ RW_2(A,130,170)\ C_2$

Since RW occurs uninterruptibly, there is no way for T_2 to interfere with T_1, and the two transactions must occur serially in one order or another.

Of course not all updates can be done this way; there are complex cases where the rows to be updated cannot be determined by a Boolean search condition, or where the amount to update is not a simple function. However this approach works well enough for most commercial application logic.

Repeatable Read (RR)[5]**.** This is the isolation level most people think is all that is meant by Serializable guarantees of Two-Phase Locking. All data items read and written have RLs and WLs taken and held long-term (until EOT). However, there are no Predicate Read Locks taken in this isolation level, so the Phantom Anomaly example at the beginning of Section 4 can occur in RR.

[5] Note that this is a confusing name. IBM (which invented Degrees of Isolation before ANSI defined Isolation Levels) has the DB2 product, in which Repeatable Read avoids the Predicate Anomaly, unlike the ANSI RR, which doesn't.

Serializable (SR). The Serializable Isolation Level takes all locks, including Data item and Predicate Read Locks and Write Locks and holds them long-term. Thus all known Anomalies are avoided by transactional histories running under SR.

Phantom Write locks held Long Term in Isolation Levels RC, RR & SR

The reason that Phantom Write locks are taken in all Isolation Levels that can perform updates (Read Uncommitted Transactions are always Read Only) is implicit in the content of the Paragraph on KVL Locking for Second Phantom Anomaly Example at the end of Section 5. Specifically, the Example says a transaction Reading through Accounts in BRANCH = 'Berkeley' to sum up the account balances must WAIT if a different, still active transaction has deleted a row (with no trace left of the record in the database). It was shown that Phantom Write locks (KVL locking) would guarantee this. It turns out that even the lowest Isolation level that can perform updates (including Deletes) of single rows must take long-term Phantom Write locks when doing so, and also must take short-term Phantom Read locks that will detect such a Phantom Write lock before Reading and so WAIT to avoid the Anomaly. The need for this is implicit in the meaning of the Isolation level name "Read Committed", since a transaction acting at this isolation level retrieving the Sum of Account balances while ignoring an outstanding delete, might get the wrong answer, and it would be doing so because it was reading uncommitted data!

Some Transactions Do Not Require SR

We have already indicated how certain banking transactions with individual row accesses for deposits, withdrawals, and transfers, could perform these actions without error at Isolation Level RC by using uninterruptible RW operations. Note too that banking applications that do not support queries of more than one row (where the row itself is locked) and do not update any columns they query through an index, will not be susceptible to Phantom Anomalies. Longer queries that banks need in the course of their business must wait for quiescent transactions.

6. Snapshot Isolation.

Snapshot Isolation (or ***SI***) is a form of concurrency control that was first presented in a 1995 paper [BBGMOO], and has since been adopted by two major Commercial Database products, Oracle and Microsoft SQL Server. We define it below.

In a DBMS System running Snapshot Isolation, the System assigns a unique timestamp when any transaction T_i starts, represented by ***Start(T_i)***, and another when T_i Commits, represented by ***Commit(T_i)***. The system is always aware of all active transactions and their Start times. The time interval [Start(T_i), Commit(T_i)] is defined to be the ***Lifetime*** of the transaction T_i. Two transactions, T_i and T_j, with overlapping Lifetimes, are said to be ***Concurrent***.

A transaction T_i running under Snapshot Isolation reads all data items from a ***Snapshot*** of data that was most recently Committed as of the time Start(T_i), except that T_i will always reread data that T_i itself has written during its Lifetime. If two concurrent Transactions T_i and T_j write the same data item A, only the first one to Commit will succeed, and the second will Abort; this is called the ***First Committer Wins*** (***FCW***) Rule.

How is a Snapshot defined? Conceptually, we pretend that all Committed data at the time transaction T_i starts has a "Snapshot" taken, so the T_i can always read data item values that were most recently committed at that time. (For a data item A that has never been updated and Committed, we pretend a *Progenitor Transaction* Committed it at System Start-up time, giving it a version named A_0.)

Of course, we don't really create a Snapshot of all the data when a transaction starts, as that would involve inefficient copying. Instead, we use the following logic. Every time a data item D is updated by transaction T_k and the transaction T_k Commits, we create a new ***Version*** of the data item called D_k; at the same time, we keep older versions of D available in the database as long as they might be referenced by some transaction. Thus we might also be keeping D_j available, a version prior to D_k that was created by transaction T_j. Note particularly that a data item version D_j is never actually *created* until T_j Commits -- up to that time, even if T_j has already performed the logic to write the data item D, it is only a *potential* new version D_j.

Now there is an order to when these versions of Data Item D were created, and the System knows when D_k is the next version after D_j; then D_k = NEXT(D_j). Given this definition, we say that any data item version D_j has valid-time-interval of timestamps in [Create(D_j), Create(NEXT(D_j))), i.e., any timestamp t such that: Create(D_j) ≤ t < Create(NEXT(D_j)). If there is no later version of D than D_j, then the time Create(NEXT(D_j)) can simply be taken to be the present. What we mean when we say that transaction T_i running under SI reads from a snapshot of data that was most recently committed as of the time Start(T_j) is this: When T_i reads the data item D, it will read D_j iff Create(D_j) ≤ Start(T_i) < Create(NEXT(D_j)).

When we speak of versions of data items, it should be understood that we refer not only to rows but to index entries as well. When a new row version is created and an index entry is added (to all indexes) to access that row version, we clearly need to differentiate index entries with the same keyvalue that access different versions of the same row. Thus the index entries themselves must have timestamps to permit the System to follow the proper row version pointer.

Snapshot Isolation Versioned Histories and Anomaly Avoidance

Snapshot Isolation can be thought of as another Isolation Level to be added to the ANSI Isolation Levels. This is appropriate because Snapshot Isolation does not guarantee true serializability. However many Anomalies that occur in lower ANSI isolation levels are avoidable with SI. We illustrate this with a number of examples, starting with the Lost Update Anomaly that can occur in Read Committed.

H1': $R_1(A,100)\ R_2(A,100)\ W_1(A,130)\ C_1\ W_2(A,140)\ C_2$

In Snapshot Isolation this Anomaly will not occur, because of the First Committer Wins (FCW) Rule.

$H1_{SI}$': $R_1(A_0,100)\ R_2(A_0,100)\ W_1(A_1,130)\ C_1\ W_2(A_2,140)\ A_2$ (because of FCW; Now Retry)
$R_3(A_1,130)\ W_3(A_3,170)\ C_3$

The Inconsistent Analysis Anomaly can occur in Read Committed; given H2',

H2': $R_1(A,100)\ R_2(B,100)\ W_2(B,50)\ R_2(A,100)\ W_2(A,150)\ R_1(B,50)\ C_1\ C_2$

None of the Reads in H2' hold locks long enough to forestall the two Writes. But in SI, we would see this:

$H2_{SI}$': $R_1(A_0,100)\ R_2(B_0,100)\ W_2(B_2,50)\ R_2(A_0,100)\ W_2(A_2,150)\ R_1(B_0,100)\ C_1\ C_2$

Since T_1 and T_2 are concurrent, T_1 cannot see any data item versions written by T_2, so T_1 sees the proper sum of A and B that existed before T_2 executed.

No transaction in Snapshot Isolation will ever see an inconsistent set of data from an intermediate transactional state.

Consider the English Language versions of the ANSI Phenomena for Isolation Levels; we will see that SI avoids them all, seemingly guaranteeing Serializability. It was this realization that gave the hint that there was a flaw in the Phenomena, since it can be shown that SI is not truly Serializable.

P1 (Dirty Read): Transaction T1 modifies a data item. Another transaction T2 then reads that data item before T1 performs a COMMIT or ROLLBACK. If T1 then performs a ROLLBACK, T2 has read a data item that was never committed and so never really existed.

Clearly this can't happen with SI, since if T2 reads a data item before T1 commits, T2 will read an earlier version than the one T1 wrote. There is no problem.

P2 (Non-Repeatable Read): Transaction T1 reads a data item. Another transaction T2 then modifies or deletes that data item and Commits. If T1 then attempts to reread the data item, it receives a modified value or discovers that the data item has been deleted.

This won't happen, because if T1 rereads the same data item a second time it will read the same version it read the first time. There is no problem.

P3 (Phantom): Transaction T1 reads a set of data items satisfying some <search cond>. Transaction T2 then creates data items that satisfy T1's <search cond> and Commits. If T1 then repeats its read with the same <search cond>, it gets a set of data items different from the first read.

As we've pointed out, SI versions all the data, including Indexes. Thus if T1 repeats its read, it will NOT find the new data items created by T2.

The reason the three Phenomena did not give Snapshot Isolation any trouble is that the assumption the Phenomena took was that if a Read by T1 follows a Write by T2, T1 will read T2's output. No thought seemed to be given to versioned histories, a surprising lack given that the intent was to apply to concurrency methods other than Locking. Versioned concurrency methods have been known for quite some time (see [BHG], Chapter 5).

To illustrate Concurrency Anomalies that can occur when running under SI we provide a few Examples..

Skew Writes SI Anomaly. Consider two data items A and B, representing balances of two bank accounts held by husband and wife, with a joint constraint that A + B > 0.

H6': $R_1(A,50)$ $R_1(B,50)$ $R_2(A,50)$ $R_2(B,50)$ $W_1(A,-40)$ $W_2(B,-40)$ C_1 C_2

Interpretation. T_1 sees A = 50 and B = 50, and concludes it can withdraw 90 from A and keep a total balance over 0, while T_2 concludes the same about withdrawing 90 from B. There is no collision on updates, so First Committer Wins doesn't prevent the problem. The values updated are "Skew", but dependent on each other. This could not happen in a Serial schedule so it is non-serializable.

Note that we could avoid such an error as this by creating a third data item C that materializes the constraint. Transactions must maintain C = A + B, and must determine that the transaction leaves C ≥ 0. Then history H6' becomes:

H6'': $R_1(A,50)$ $R_1(B,50)$ $R_1(C,100)$ $R_2(A,50)$ $R_2(B,50)$ $R_2(C,100)$ $W_1(A,-40)$ $W_1(C,10)$ C_1 $W_2(B,-40)$ $W_2(A,10)$ A_2 (Because FCW on C update; a retry of T_2 will fail because of too small a balance.)

A second way to prevent this Anomaly in H6' is to require T_1 and T_2 to Read both A and B FOR UPDATE. (There is a way to Select For Update in SQL.) Each will actually update one data item but reads the other for update so no other transaction will be able to change the value being depended on -- such an attempt to change will result in FCW.

Here is a different Anomaly, also dependent on common constraints, but which has the earmarks of a Phantom Anomaly.

Phantom SI Anomaly. We have a table named employees, a table named projects, and a table named assignments that lists assignments for given employees to specific projects on a given day for an integer number of hours.

employees		projects		assignments			
eid	ename	prid	projname	eid	prid	date	no_hrs
e01	Smith	P01	Bridge	e01	P01	11/22/98	2

Assume multiple rows are given. There is a constraint that SUM(no_hrs) for any eid on any date must not exceed 8. Think how we would execute the following:

```
exec sql select sum(no_hrs) into :tothrs from assignments
   where eid = :eid and date = :date;

if (tothrs + newhrs <= 8)
   exec sql insert into assignment values (:eid, :prid, :date, :newhrs);
```

But assume, as in H6', that two concurrent transactions read this set of values, then both insert new assignments rows with no_hrs = 2. The total no_hrs in each case could have started at 6, and this means that the two transactions together have caused the total no_hrs to be updated to 10.

This is not a "Skew" Write. Each insert is totally new, but depended on a sum of no_hrs that would be different if the other insert were known to it. There is a way to guard against this sort of Anomaly as well, by materializing the constraint (a common solution). We define another table: totassgn, with columns eid, date, and tot_no_hrs. If the tot_no_hrs is kept up to date in each transaction making a new assignment for a given eid on a given date, no pair of assignment transactions will be able to cause the Anomaly displayed without causing a FCW error in tot_no_hrs.

So Snapshot Isolation does NOT imply Serializability. However it is quite popular with database practitioners, since it seems that Queries will be able to provide consistent results without ever having to WAIT for update transactions. The query values might be slightly old but they would be even older if WAITs were enforced, and we can show that all Queries are reading commit-consistent data as of a recent period. There is no comparable way to perform non-waiting queries in Locking schedulers, so Snapshot Isolation has an advantage. A short article in SIGMOD record demonstrated a subtle Anomaly that could occur with a query in Snapshot isolation, but it is probably not a practical concern for most users, so Snapshot Isolation is much sought after.

References

[AGRA] R. Agrawal, M. Carey, M. Livny, "*Concurrency Control Performance Modeling: Alternatives and Implications*", ACM Trans. on Database Systems, 12(4), Dec,1987.

[ANSI] ANSI INCITS 135-1992 (R1998) Information Systems - Database Language - SQL; INCITS/ISO/IEC 9075-2-1999 Information Technology- Database Language-SQL Part 2: Foundation (SQL/Foundation); INCITS/ISO/IEC 9075-2-2003 (foundations part)-Information technology - Database languages - SQL - Part 2: Foundation(SQL/Foundation) http://webstore.ansi.org/ansidocstore/

[BACH] Charles W. Bachman, Private Communication.

[BBGMOO] H. Berenson,. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A Critique of ANSI SQL Isolation Levels," ACM SIGMOD, May 1995, pp. 1-10. URL: http://www.cs.duke.edu/~junyang/courses/cps216-2003-spring/papers/berenson-etal-1995.pdf

[BHG] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison Wesley, 1987. (The text is out of print but can be downloaded from http://research.microsoft.com/pubs/ccontrol/default.htm)

[CHETAL81] D. Chamberlain et al. A History and Evaluation of System R. CACM V24.10, pp. 632-646, Oct. 1981. (This paper is also in: Michael Stonebraker and Joseph Hellerstein, Readings in Database Systems, Third Edition, Morgan Kaufmann 1998.)

[DEWITT] D. DeWitt et Al., "Implementation Techniques for Main Memory Database Systems," Proc. ACM SIGMOD Conf, June 1984.

[EGLT] K. P. Eswaran, J. Gray, R. Lorie, I. Traiger, *"The Notions of Consistency and Predicate Locks in a Database System,"* CACM V19.11, pp. 624-633, Nov. 1976.

[FOO} A. Fekete, E. O'Neil, P. O'Neil, *"A Read-Only Transaction Anomaly Under Snapshot Isolation,"* ACM SIGMOD Record, Vol. 33, No. 3, Sept. 2004

[GLP] J. Gray, R. Lorie, and F. Putzolu, *"Granularity of Locks in a Large Shared Data Base,"* Proc. First Conf on Very Large Databases, 1975

[GLPT] J. Gray, R. Lorie, F. Putzolu, and I. Traiger, *"Granularity of Locks and Degrees of Consistency in a Shared Data Base,"* Published in 1976, now available in "Readings in Database Systems", Fourth Edition, Chapter 4, J. M. Hellerstein and M. Stonebraker, Eds., M.I.T. Press 2005.

[GR97] Jim Gray and Andreas Reuter. "Transaction Processing: Concepts and Techniques, 3rd Printing. Morgan Kaufmann, 1997.

[GRAY] Jim Gray, Private Communication.

[GRAY81] Jim Gray. "The transaction concept: Virtues and limitations" *Proceedings of the 7th International Conference on Very Large Data Bases*, Sept. 1981. pages 144—154.

[GRAYPUT] J. Gray and G. F. Putzolu, "The Five-minute Rule for Trading Memory for Disc Accesses, and the 10 Byte Rule for Trading Memory for CPU Time," Proceedings of SIGMOD 87, June 1987, pp. 395-398.

[LRU-K] E. O'Neil, P. O'Neil, G, Weikum, "The LRU-K Page-Replacement Algorithm for Database Disk Buffering," ACM SIGMOD Conference, May 1993, Washington, D.C, Proceedings pp. 296-306.

[MOHAN90] C. Mohan. Aries/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. Proceedings of the 16th VLDB Conference, 1990, pp. 392-405.

[MOHAN92] C. Mohan. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. Proceedings of the ACM SIGMOD International Conference on Management of Data, June 1992, pp. 371-380.

[Sabre] Wikipedia: http://en.wikipedia.org/wiki/Sabre_(computer_system) Sabre Computer System: History

[TAYGS] Y.C. Tay, R. Suri, and N. Goodman, *"Locking Performance in Centralized Databases,"* ACM Trans. on Database Systems 10(4), pp 415-462, December 1985.

[TPC-A,TPC-B,TPC-C] The Benchmark Handbook for Database and Transaction Processing Systems, Second Edition: Final Chapters. Jim Gray, Editor, Morgan Kaufmann 1993. http://www.sigmod.org/dblp/db/books/collections/gray93.html