Lecture Notes for Database Systems

Patrick E. O'Neil

Chapter 10

Class 20.

We have encountered the idea of a transaction before in Embedded SQL.

Def. 10.1 Transaction. A transaction is a means to package together a number of database operations performed by a process, so the database system can provide several guarantees, called the ACID properties.

Think of writing: BEGIN TRANSACTION op1 op2 . . . opN END TRANSACTION

Then all ops within the transaction are packaged together.

There is no actual BEGIN TRANSACTION statement in SQL. A transaction is begun by a system when there is none in progress and the application first performs an operation that accesses data: Select, Insert, Update, etc.

The application logic can end a transaction successfully by executing:

```
   exec sql commit work;       /* called simply a Commit                    * /
```

Then any updates performed by operations in the transaction are successfully completed and made permanent and all locks on data items are released. Alternatively:

```
   exec sql rollback work;     /* called an Abort                           * /
```

means that the transaction was unsuccessful: all updates are reversed, and locks on data items are released.

The ACID guarantees are extremely important -- This and SQL is what differentiates a database from a file system.

Imagine that you are trying to do banking applications on the UNIX file system (which has it's own buffers, but no transactions). There will be a number of problems, the kind that faced database practitioners in the 50s.

1. **Inconsistent result**. Our application is transferring money from one account to another (different pages). One account balance gets out to disk (run out of buffer space) and then the computer crashes.

When bring computer up again, have no idea what used to be in memory buffer, and on disk we have destroyed money.

2. **Errors of concurrent execution**. (One kind: Inconsistent Analysis.) Teller 1 transfers money from Acct A to Acct B of the same customer, while Teller 2 is performing a credit check by adding balances of A and B. Teller 2 can see A after transfer subtracted, B before transfer added.

3. **Uncertainty as to when changes become permanent**. At the very least, we want to know when it is safe to hand out money: don't want to forget we did it if system crashes, then only data on disk is safe.

Want this to happen at transaction commit. And don't want to have to write out all rows involved in transaction (teller cash balance -- very popular, we buffer it to save reads and want to save writes as well).

To solve these problems, systems analysts came up with idea of transaction (formalized in 1970s). Here are ACID guarantees:

Atomicity. The set of record updates that are part of a transaction are indivisible (either they all happen or none happen). This is true even in presence of a crash (see Durability, below).

Consistency. If all the individual processes follow certain rules (money is neither created nor destroyed) and use transactions right, then the rules won't be broken by any set of transactions acting together. Implied by Isolation, below.

Isolation. Means that operations of different transactions seem not to be interleaved in time -- as if ALL operations of one Tx before or after all operations of any other Tx.

Durability. When the system returns to the logic after a Commit Work statement, it guarantees that all Tx Updates are on disk. Now ATM machine can hand out money.

The system is kind of clever about Durability. It doesn't want to force all updated disk pages out of buffer onto disk with each Tx Commit.

So it writes a set of notes to itself on disk (called logs). After crash run *Recovery* (also called *Restart*) and makes sure notes translate into appropriate updates.

What about Read-Only Tx? (No data updates, only Selects.) Atomicity and Durability have no effect, but Isolation does.

Money spent on Transactional systems today is about SIX BILLION DOLLARS A YEAR. We're being rigorous about some of this for a BUSINESS reason.

10.1 Transactional Histories.

Reads and Writes of data items. A data item might be a row of a table or it might be an index entry or set of entries. For now talking about rows.

Read a data item when access it without changing it. Often a select.

```
   select val into :pgmval1 from T1 where uniqueid = A;
```

We will write this as $R_i(A)$: transaction with identification number i reads data item A. Kind of rough — won't always have be retrieving by uniqueid = A. But it means that we are reading a row identified as A. Now:

```
   update T1 set val = pgmval2 where uniqueid = B;
```

we will write this as $W_j(B)$; Tx j writes B; say Update results in Write.

Can get complicated. Really reading an index entry as well to write B. Consider:

```
   update T1 set val = val + 2 where uniqueid = B;
```

Have to read an index entry, R_j (predicate: uniqueid = B), then a pair of row operations: $R_j(B)$ (have to read it first, then update it) $W_j(B)$. Have to read it in this case before can write it.

```
   update T set val = val + 2 where uniqueid between :low and :high;
```

Will result in a lot of operations: R_j(predicate: uniqueid between :low and :high), then $R_j(B1)$ $W_j(B1)$ $R_j(B2)$ $W_j(B2)$ . . . $R_j(BN)$ $W_j(BN)$.

>>The reason for this notation is that often have to consider complex interleaved histories of concurrent transactions; Example history:

(10.1.2) . . . $R_2(A)$ $W_2(A)$ $R_1(A)$ $R_1(B)$ $R_2(B)$ $W_2(B)$ C_1 C_2 . . .

Note C_i means commit by Tx i. A sequence of operations like this is known as a *History* or sometimes a *Schedule*.

A history results from a series of operations submitted by users, translated into R & W operations at the level of the Scheduler. See Fig. 10.1.

It is the job of the scheduler to look at the history of operations as it comes in and provide the Isolation guarantee, by sometimes delaying some operations, and occasionally insisting that some transactions be aborted.

By this means it assures that the sequence of operations is equivalent in effect to some serial schedule (all ops of a Tx are performed in sequence with no interleaving with other transactions). See Figure 10.1, pg. 640.

In fact, (10.1.2) above is an ILLEGAL schedule. Because we can THINK of a situation where this sequence of operations gives an inconsistent result.

Example 10.1.1. Say that the two elements A and B in (10.1.2) are Acct records with each having balance 50 to begin with. Inconsistent Analysis.

T_1 is adding up balances of two accounts, T_2 is transferring 30 units from A to B.

. . . $R_2(A, 50)$ $W_2(A, 20)$ $R_1(A, 20)$ $R_1(B, 50)$ $R_2(B, 50)$ $W_2(B, 80)$ C_1 C_2 . . .

And T determines that the customer fails the credit check (because under balance total of 80, say).

But this could never have happened in a serial schedule, where all operation of T_2 occurred before or after all operations of T_2.

. . . $R_2(A, 50)$ $W_2(A, 20)$ $R_2(B, 50)$ $W_2(B, 80)$ C_2 $R_1(A, 20)$ $R_1(B, 80)$ C_2 . . .
or
. . . $R_1(A, 50)$ $R_1(B, 50)$ C_1 $R_2(A, 50)$ $W_2(A, 20)$ $R_2(B, 50)$ $W_2(B, 80)$ C_2 . . .

And in both cases, T_1 sees total of 100, a Consistent View.

Notice we INTERPRETED the Reads and Writes of (10.1.2) to create a model of what was being read and written to show there was an inconsistency.

This would not be done by the Scheduler. It simply follows a number of rules we explain shortly. We maintain that a serial history is always consistent under any interpretation.

10.2. Interleaved Read/Write Operations

Quick>>
If a serial history is always consistent, why don't we just enforce serial histories.

The scheduler could take the first operation that it encounters of a given transaction (T_2 in the above example) and delay all ops of other Txs (the Scheduler is allowed to do this) until all operations of T_2 are completed and the transaction commits (C_2).

Reason we don't do this? Performance. It turns out that an average Tx has relatively small CPU bursts and then I/O during which CPU has nothing to do. See Fig 10.3, pg. 644. When I/O is complete, CPU can start up again.

What do we want to do? Let another Tx run (call this another thread) during slack CPU time. (Interleave). Doesn't help much if have only one disk (disk is bottleneck). See Fig 10.4, pg. 644.

But if we have two disks in use all the time we get about twice the throughput. Fig 10.5, pg. 645.

And if we have many disks in use, we can keep the CPU 100% occupied. Fig 10.6, pg 646.

In actuality, everything doesn't work out perfectly evenly as in Fig 10.6. Have multiple threads and multiple disks, and like throwing darts at slots.

Try to have enough threads running to keep lots of disk occupied so CPU is 90% occupied. When one thread does an I/O, want to find another thread with completed I/O ready to run again.

Leave this to you -- covered in Homework.

10.3 Serializability and the Precedence Graph.

We want to come up with a set of rules for the Scheduler to allow operations by interleaved transactions and guarantee *Serializability*.

Serializability means the series of operations is EQUIVALENT to a Serial schedule, where operations of Tx are not interleaved.

How can we guarantee this? First notice that if two transactions never access the same data items, it doesn't matter that they're interleaved.

We can commute ops in the history of requests permitted by the scheduler until all ops of one Tx are together (serial history). The operations don't affect each other, and order doesn't matter.

We say that the Scheduler is reading a history H (order operations are submitted) and is going to create a serializable history S(H) (by delay, etc.) where operations can be commuted to a serial history.

OK, now if we have operations by two different transactions that do affect the same data item, what then?

There are only four possibilities: R or W by T_1 followed by R or W by T_2. Consider history:

$. . . R_1(A) . . . W_2(A) . . .$

Would it matter if the order were reversed? YES. Can easily imagine an interpretation where T_2 changes data T_1 reads: if T_1 reads it first, sees old version, if reads it after T_2 changes it, sees later version.

We use the notation:

$R_1(A) <<_H W_2(A)$

to mean that $R_1(A)$ comes before $W_2(A)$ in H, and what we have just noticed is that whenever we have the ordering in H we must also have:

$R_1(A) <<_{S(H)} W_2(A)$

That is, these ops must occur in the same order in the serializable schedule put out by the Scheduler. If $R_1(A) <<_H W_2(A)$ then $R_1(A) <<_H W_2(A)$.

Now these transaction numbers are just arbitrarily assigned labels, so it is clear we could have written the above as follows:

If $R_2(A) <<_H W_1(A)$ then $R_2(A) <<_H W_1(A)$.

Here Tx 1 and Tx 2 have exchanged labels. This is another one of the four cases. Now what can we say about the following?

$R_1(A) <<_H R_2(A)$

This can be commuted -- reads can come in any order since they don't affect each other. Note that if there is a third transaction, T_3, where:

$R_1(A) <<_H W_3(A) <<_H R_2(A)$

Then the reads cannot be commuted (because we cannot commute either one with $W_3(A)$), but this is because of application of the earlier rules, not depending on the reads as they affect each other.

Finally, we consider:

$W_1(A) <<_H W_2(A)$

And it should be clear that these two operations cannot commute. The ultimate outcome of the value of A would change. That is:

If $W_1(A) <<_H W_2(A)$ then $W_1(A) <<_{S(H)} W_2(A)$

To summarize our discussion, we have Definition 10.3.1, pg. 650.

Def. 10.3.1. Two operations $X_i(A)$ and $Y_j(B)$ in a history are said to *conflict* (i.e., the order matters) if and only if the following three conditions hold:

(1) $A \equiv B$. Operations on distinct data items never conflict.

(2) $i \neq j$. Operations conflict only if they are performed by different Txs.

(3) One of the two operations X or Y is a write, W. (Other can be R or W.)

Note in connection with (2) that two operations of the SAME transaction also cannot be commuted in a history, but not because they conflict. If the scheduler delays the first, the second one will not be submitted.

Class 21.

We have just defined the idea of two conflicting operations. (Repeat?)

We shall now show how some histories can be shown not not to be serial-izable. Then we show that such histories can be characterized by an easily identified characteristic in terms of conflicting operations.

To show that a history is not serializable (SR), we use an *interpretation* of the history.

Def. 10.3.2. An interpretation of an arbitrary history H consists of 3 parts. (1) A description of the purpose of the logic being performed. (2) Spec-ification of precise values for data items being read and written in the history. (3) A consistency rule, a property that is obviously preserved by isolated transactions of the logic defined in (1).

Ex. 10.3.1. Example 10.3.1. Here is a history, H1, we clain is not SR.

$H1 = R_2(A)\ W_2(A)\ R_1(A)\ R_1(B)\ R_2(B)\ W_2(B)\ C_1\ C_2$

Here is an interpretation. T_1 is doing a credit check, adding up the balances of A and B. T is transferring money from A to B. Here is the consistency rule: Neither transaction creates or destroys money. Values for H1 are:

$H1' = R_2(A,50)\ W_2(A,20)\ R_1(A,20)\ R_1(B,50)\ R_2(B,50)\ W_2(B,70)\ C_1\ C_2$

The schedule H1 is not SR because H1' shows an *inconsistent result:* sum of 70 for balances A and B, though no money was destroyed by T_2 in the transfer from A to B. This could not have occurred in a serial execution.

The concept of conflicting operations gives us a direct way to confirm that the history H1 is not SR. Note the second and third operations of H1, $W_2(A)$ and $R_1(A)$. Since W(A) comes before R(A) in H1, written:

$W_2(A) <<_{H1} R_1(A)$

We know since these operations conflict that they must occur in the same order in any equivalent serial history, S(H1), i.e.: $W_2(A) <<_{S(H1)} R_1(A)$

Now in a serial history, all operations of a transaction occur together

Thus $W_2(A) <<_{S(H1)} R_1(A)$ means that $T_2 <<_{S(H1)} T_1$, i.e. T_2 occurs before T_1 in any serial history S(H1) (there might be more than one).

But now consider the fourth and sixth operations of H1. We have:

$R_1(B) <<_{H1} W_2(B)$

Since these operations conflict, we also have $R_1(B) <<_{S(H1)} W_2(B)$

But this implies that T_1 comes before T_2 , $T_1 <<_{S(H1)} T_2$, in any equivalent serial history H1. And this is at odds with our previous conclusion.

In any serial history S(H1), either $T_1 <<_{S(H1)} T_2$ or $T_2 <<_{S(H1)} T_1$, not both. Since we conclude from examining H1 that both occur, S(H1) must not really exist. Therefore, H1 is not SR.

We illustrate this technique a few more times, and then prove a general characterization of SR histories in terms of conflicting operations.

Ex. 10.3.2. Consider the history:

$H2 = R_1(A)\ R_2(A)\ W_1(A)\ W_2(A)\ C_1\ C_2$

We give a interpretation of this history as a paradigm called a *lost update*.

Assume that A is a bank balance starting with the value 100 and T1 tries to add 40 to the balance at the same time that T2 tries to add 50.

$H2' = R_1(A, 100)\ R_2(A, 100)\ W_1(A, 140)\ W_2(A, 150)\ C_1\ C_2$

Clearly the final result is 150, and we have lost the update where we added 40. This couldn't happen in a serial schedule, so H1 is non-SR.

In terms of conflicting operations, note that operations 1 and 4 imply that $T1 <<_{S(H2)} T2$. But operations 2 and 3 imply that $T2 <<_{S(H2)} T1$. No SR schedule could have both these properties, therefore H2 is non-SR.

By the way, this example illustrates that a conflicting pair of the form $R_1(A)$ . . . $W_2(A)$ does indeed impose an order on the transactions, T1 << T2, in any equivalent serial history.

H2 has no other types of pairs that COULD conflict and make H2 non-SR.

Ex 10.3.3. Consider the history:

$H3 = W_1(A)\ W_2(A)\ W_2(B)\ W_1(B)\ C_1\ C_2$

This example will illustrate that a conflicting pair $W_1(A) \ldots W_2(A)$ can impose an order on the transactions T1 << T2 in any equivalent SR history.

Understand that these are what are known as "Blind Writes": there are no Reads at all involved in the transactions.

Assume the logic of the program is that T1 and T2 are both meant to "top up" the two accounts A and B, setting the sum of the balances to 100.

T1 does this by setting A and B both to 50, T2 does it by setting A to 80 and B to 20. Here is the result for the interleaved history H3.

$H3' = W_1(A, 50)\ W_2(A, 80)\ W_2(B, 80)\ W_1(B, 50)\ C_1\ C_2$

Clearly in any serial execution, the result would have A + B = 100. But with H3' the end value for A is 80 and for B is 50.

To show that H3 is non-SR by using conflicting operations, note that operations 1 and 2 imply T1 << T2, and operations 3 and 4 that T2 << T1.

Seemingly, the argument that an interleaved history H is non-SR seems to reduce to looking at conflicting pairs of operations and keeping track of the order in which the transactions will occur in an equivalent S(H).

When there are two transactions, T1 and T2, we expect to find in a non-SR schedule that $T1 <<_{S(H)} T2$ and $T2 <<_{S(H)} T1$, an impossibility.

If we don't have such an impossibility arise from conflicting operations in a history H, does that mean that H is SR?

And what about histories with 3 or more transactions involved? WIll we ever see something impossible other than $T1 <<_{S(H)} T2$ and $T2 <<_{S(H)} T1$?

We start by defining a Precedence Graph. The idea here is that this allows us to track all conflicting pairs of operations in a history H.

Def. 10.3.3. The Precedence Graph. A precedence graph for a history H is a directed graph denoted by PG(H).

The vertices of PG(H) correspond to the transactions that have COMMITTED in H: that is, transaction Ti where C exists as an operation in H.

An edge Ti -> Tj exists in PG(H) whenever two conflicting operations X_i and Y_j occur in that order in H. Thus, Ti -> Tj should be interpreted to mean that Ti must precede Tj in any equivalent serial history S(H).

Whenever a pair of operations conflict in H for COMMITTED transactions, we can draw the corresponding direct arc in the Precedence Graph, PG(H).

The reason uncommitted transactions don't count is that we're trying to figure out what the scheduler can allow. Uncommitted transactions can always be aborted, and then it will be as if they didn't happen.

It would be unfair to hold uncommitted transactions against the scheduler by saying the history is non-SR because of them.

The examples we have given above of impossible conditions arising from conflicting operations look like this:

```
T1 ----------------------> T2
 ^<-----------------------/
```

Of course this is what is called a circuit in a directed graph (a digraph). This suggests other problems that could arise with 3 or more Txs.

It should be clear that if PG(H) has a circuit, there is no way to put the transactions in serial order so Ti always comes befor Tj for all edges Ti -> Tj in the circuit.

There'll always be one edge "pointing backward" in time, and that's a contradiction, since Ti -> Tj means Ti should come BEFORE Tj in S(H).

How do we make this intuition rigorous? And if PG(H) doesn't have a circuit, does that mean the history is SR?

Thm. 10.3.4. **The Serializability Theorem**. A history H has an equivalent serial execution S(H) iff the precedence graph PG(H) contains no circuit.

Proof. Leave only if for exercises at end of chapter. I.e., will show there that a circuit in PG(H) implies there is no serial ordering of transactions.

Here we prove that if PG(H) contains no circuit, there is a serial ordering of the transactions so no edge of PG(H) ever points from a later to an earlier transaction.

Assume there are m transactions involved, and label them T1, T2, . . ., Tm.

We are trying to find a reordering of the integers 1 to m, i(1), i(2), . . ., i(m), so that Ti(1), Ti(2), . . ., Ti(m) is the desired serial schedule.

Assume a lemma to prove later: In any directed graph with no circuit there is always a vertex with no edge entering it.

OK, so we are assuming PG(H) has no circuit, and thus there is a vertex, or transaction, Tk, with no edge entering it. We choose Tk to be Ti(1).

Note that since Ti(1) has no edge entering it, there is no conflict in H that forces some other transaction to come earlier.

(This fits our intuition perfectly. All other transactions can be placed after it in time, and there won't be an edge going backward in time.)

Now remove this vertex, Ti(1), from PG(H) and all edges leaving it. Call the resulting graph $PG^1(H)$.

By the Lemma, there is now a vertex in $PG^1(H)$ with no edge entering it. Call that vertex Ti(2).

(Note that an edge from Ti(1) might enter Ti(2), but that edge doesn't count because it's been removed from $PG^1(H)$.)

Continue in this fashion, removing Ti(2) and all it's edges to form $PG^2(H)$, and so on, choosing Ti(3) from $PG^2(H)$, . . ., Ti(m) from $PG^{m-1}(H)$.

By construction, no edge of PG(H) will ever point backward in the sequence S(H), from Ti(m) to Ti(n), m > n.

The algorithm we have used to determine this sequence is known as a *topological sort*. This was a hiring question I saw asked at Microsoft.

The proof is complete, and we now know how to create an equivalen SR schedule from a history whose precedence graph has no circuit.

Lemma 10.3.5. In any finite directed acyclic graph G there is always a vertex with no edges entering it.

Proof. Choose any vertex v1 from G. Either this has the desired property, or there is an edge entering it from another vertex v2.

(There might be several edges entering v1, but choose one.)

Now v2 either has the desired property or there is an edge entering it from vertex v3. We continue in this way, and either the sequence stops at some vertex vm, or the sequence continues forever.

If the sequence stops at a vertex vm, that's because there is no edge entering vm, and we have found the desired vertex.

But if the sequence continues forever, since this is a finite graph, sooner or later in the sequence we will have to have a repeated vertex.

Say that when we add vertex vn, it is the same vertex as some previously mentioned vertex in the sequence, vi.

Then there is a path from vn -> v(n-1) -> . . . v(i+1) ->vi, where vi ≡ vn. But this is the definition of a circuit, which we said was impossible.

Therefore the sequence had to terminate with vm and that vertex was the one desired with no edges entering.

Class 22.

10.4 Locking Ensures Serializabilty

See Fig. 10.8, pg. 609. TM passes on calls such as fetch, select, insert, delete, abort; Scheduler interprets them as: $R_i(A)$, $W_j(B)$.

It is the job of the scheduler to make sure that no non-SR schedules get past. This is normally done with *Two-Phase Locking*, or 2PL.

Def. 10.4.1. 2PL. Locks taken in released following three rules.

(1) Before Tx i can read a data item, $R_i(A)$, scheduler attempts to Read Lock the item on it's behalf, $RL_i(A)$; before $W_i(A)$, try Write Lock, $WL_i(A)$.

(2) If conflicting lock on item exists, requesting Tx must WAIT. (Conflicting locks corresponding to conflicting ops: two locks on a data item conflict if they are attempted by different Txs and at least one of them is a WL).

(3) There are two phases to locking, the growing phase and the shrinking phase (when locks are released: $RU_i(A)$; The scheduler must ensure that can't shrink (drop a lock) and then grow again (take a new lock).

Rule (3) implies can release locks before Commit; More usual to release all locks at once on Commit, and we shall assume this in what follows.

Note that <u>a transaction can never conflict with its own locks!</u> If Ti holds RL on A, can get WL so long as no other Tx holds a lock (must be RL).

A Tx with a WL doesn't need a RL (WL more powerful than RL).

Clearly locking is defined to guarantee that a circuit in the Precedence Graph can never occur. The first Tx to lock an item forces any other Tx that gets to it second to "come later" in any SG.

But what if other Tx already holds a lock the first one now needs? This would mean a circuit, but in the WAIT rules of locking it means NEITHER Tx CAN EVER GO FORWARD AGAIN. This is a DEADLOCK. Example shortly.

Side effect of 2PL is that Deadlocks can occur: When a deadlock occurs, scheduler will recognize it and force one of the Txs involved to Abort. (Note, there might be more than 2 Txs involved in a Deadlock.)

Ex. Here is history not SR (Error in text: this is a variant of Ex. 10.4.1).

H4 = $R_1(A)$ $R_2(A)$ $W_2(A)$ $R_2(B)$ $W_2(B)$ $R_1(B)$ C_1 C_2

Same idea as 10.3.1 why it is non-SR: T2 reads two balances that start out A=50 and B=50, T1 moves 30 from A to B. Non-SR history because T1 sees A=50 and B=80. Now try locking and releasing locks at commit.

$RL_1(A)$ $R_1(A)$ $RL_2(A)$ $R_2(A)$ $WL_2(A)$ (conflicting lock held by T1 so T2 must WAIT) $RL_1(B)$ $R_1(B)$ C_1 (now T2 can get $WL_2(A)$) $W_2(A)$ $RL_2(B)$ $R_2(B)$ $WL_2(B)$ $W_2(B)$ C_2

Works fine: T1 now sees A=50, B=50. Serial schedule, T1 then T2.

But what if allowed to Unlock and then acquire more locks later. Get non-SR schedule. Shows necessity of 2PL Rule (3).

$RL_1(A)$ $R_1(A)$ $RU_1(A)$ $RL_2(A)$ $R_2(A)$ $WL_2(A)$ $W_2(A)$ $WU_2(A)$ $RL_2(B)$ $R_2(B)$ $WL_2(B)$ $W_2(B)$ $WU_2(B)$ $RL_1(B)$ $R_1(B)$ C_1 C_2

So H4 above is possible. But only 2PL rule broken is that T1 and T2 unlock rows, then lock other rows later.

The Waits-For Graph. How scheduler checks if deadlock occurs. Vertices are currently active Txs, Directed edgs Ti -> Tj iff Ti is waiting for a lock held by Tj.

(Note, might be waiting for lock held by several other Txs. And possibly get in queue for W lock behind others who are also waiting. Draw picture.)

The scheduler performs lock operations and if Tx required to wait, draws new directed edges resulting, then checks for circuit.

Ex 10.4.2. Here is schedule like H4 above, where T2 reverses order it touches A and B (now touches B first), but same example shows non-SR.

H5 = $R_1(A)$ $R_2(B)$ $W_2(B)$ $R_2(A)$ $W_2(A)$ $R_1(B)$ C_1 C_2

Locking result:

$RL_1(A)$ $R_1(A)$ $RL_2(B)$ $R_2(B)$ $WL_2(B)$ $W_2(B)$ $RL_2(A)$ $R_2(A)$ $WL_2(A)$ (Fails: RL1(A) held, T2 must WAIT for T1 to complete and release locks) $RL_1(B)$ (Fails: WL2(B) held, T1 must wait for T2 to complete: But this is a deadlock! Choose T2 as victim (T1 chosen in text)) A2 (now $RL_1(B)$ will succeed) R1(B) C1 (start T2 over, retry, it gets Tx number 3) $RL_3(B)$ $R_3(B)$ $WL_3(B)$ $W_3(B)$ $RL_3(A)$ $R_3(A)$ $WL_3(A)$ W3(A) C3.

Locking serialized T1, then T2 (retried as T3).

Thm. 10.4.2. Locking Theorem. A history of transactional operations that follows the 2PL discipline is SR.

First, **Lemma 10.4.3**. If H is a Locking Extended History that is 2PL and the edge Ti -> Tj is in PG(H), then there must exist a data item D and two conflicting operations Xi(D) and Yj(D) such that XUi(D) <<H YLj(D).

Proof. Since Ti -> Tj in PG(H), there must be two conflicting ops Xi(D) and Yj(D) such that Xi(D) <<H Yj(D).

By the definition of 2PL, there must be locking and unlocking ops on either side of both ops, e.g.: XLi(D) <<H Xi(D) <<H XUi(D).

Now between the lock and unlock for Xi(D), the X lock is held by Ti and similarly for Yj(D) and Tj. Since X and Y conflict, the locks conflict and the intervals cannot overlap. Thus, since Xi(D) <<H Yj(D), we must have:

XLi(D) <<H Xi(D) <<H XUi(D) <<H YLj(D) <<H Yj(D) <<H YUj(D)

And in particular XUi(D) <<H YLj(D).

Proof of Thm. 10.4.2. We want to show that every 2PL history H is SR.

Assume in contradiction that there is a cycle T1 -> T2 -> . . . -> Tn -> T1 in PG(H). By the Lemma, for each pair Tk -> T(k+1), there is a data item Dk where XUk(Dk) <<H YL(k+1)(Dk). We write this out as follows:

1. XU1(D1) <<H YL2(D1)
2. XU2(D2) <<H YL3(D2)
   . . .
n-1. XU(n-1)(D(n-1)) <<H YLn(D(n-1))
n. XUn(Dn) <<H YL1(Dn) (Note, T1 is T(n+1) too.)

But now have (in 1.) an unlock of a data item by T1 before (in n.) a lock of a data item. So not 2PL after all. Contradiction.

Therefore H is 2PL implies no circuit in the PG(H), and thus H is SR.

Note that not all SR schedules would obey 2PL. E.g., the following is SR:

H7 = R1(A) W1(A) R2(A) R1(B) W1(B) R2(B) C1 C2

But it is not 2PL (T2 breaks through locks held by T1). We can optimistically allow a Tx to break through locks in the hopes that a circuit won't
occur in PG(H). But most databases don't do that.

Class 23.

10.5 Levels of Isolation

The idea of Isolation Levels, defined in ANSI SQL-92, is that people might want to gain more concurrency, even at the expense of imperfect isolation.

A paper by Tay showed that when there is serious loss of throughput due to locking, it is generally not because of deadlock aborts (having to retry) but simply because of transactions being *blocked* and having to wait.

Recall that the reason for interleaving transaction operations, rather than just insisting on serial schedules, was so we could keep the CPU busy.

We want there to always be a new transaction to run when the running transaction did an I/O wait.

But if we assume that a lot of transactions are waiting for locks, we lose this. There might be only one transaction running even if we have 20 trying to run. All but one of the transactions are in a wait chain!

So the idea is to be less strict about locking and let more transactions run. The problem is that dropping proper 2PL might cause SERIOUS errors in applications. But people STILL do it.

The idea behind ANSI SQL-99 Isolation Levels is to weaken how locks are held. Locks aren't always taken, and even when they are, many locks are released before EOT.

And more locks are taken after some locks are released in these schemes. Not Two-Phase, so not perfect Isolation.

(Note in passing, that ANSI SQL-92 was originally intended to define isolation levels that did not require locking, but it has been shown that the definitions failed to do this. Thus the locking interpretation is right.)

Define *short-term locks* to mean a lock is taken prior to the 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 (Fig. 10.9 -- some difference from the text):

	Write locks on rows of a table are long term	Read Locks on rows of a table are long term	Read locks on predicates are 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 Write Predicate Locks are taken and held long-term in all isolation levels listed. What this means is explained later.

In Read Uncommitted (RU), no Read locks are taken, thus can read data on which Write lock exists (nothing to stop you if don't have to WAIT for RL).

Thus can read uncommitted data; it will be wrong if Tx that changed it later aborts. But RU is just to get a STATISTICAL idea of sales during the day (say). CEO wants to know ballpark figure -- OK if not exact.

In Read Committed (RC), we take Write locks and hold them to EOT, and Read Locks on rows read and predicates and release immediately. (Cover predicates below.)

Problem that can arise is serious one, Lost Update (Example 10.3.2):

. . . $R_1(A,100)$ $R_2(A,100)$ $W_1(A,140)$ $W_2(A,150)$ C_1 C_2 **. . .**

Since R locks are released immediately, nothing stops the later Writes, and the increment of 40 is overwritten by an increment of 50, instead of the two increments adding to give 90.

Call this the Scholar's Lost Update Anomaly (since many people say Lost Update only happens at Read Uncommitted).

This is EXTREMELY serious, obviously, and an example of lost update in SQL is given in Figure 10.12 (pg. 666) for a slightly more restrictive level: Cursor Stability. Applications that use RC must avoid this kind of update.

In Figure 10.11, we see how to avoid this by doing the Update indivisibly in a single operation.

Not all Updates can be done this way, however, because of complex cases where the rows to bue updated cannot be determined by a Boolean search condition, or where the amount to update is not a simple function.

It turns out the IBM's Cursor Stability guarantees a special lock will be held on current row under cursor, and at first it was thought that ANSI Read Committed guaranteed that, but it does not.

Probably most products implement a lock on current of cursor row, but there is no guarantee. NEED TO TEST if going to depend on this.

In Repeatable Read Isolation, this is the isolation level that most people think is all that is meand by 2PL. All data items read and written have RLs and WLs taken and held long-term, until EOT.

So what's wrong? What can happen?

Example 10.5.3, pg. 666, Phantom Update Anomaly.

R_1(predicate: branch_id = 'SFBay') R_1(A1,100.00) R_1(A2, 100.00)
R_1(A3,100.00) I_2(A4, branch_id = 'SFBay', balance =100.00)
R_2(branch_totals, branch_id = SFBay, 300.00)
W_2(branch_totals, branch_id = SFBay ,400.00) C_2
R_1(branch_totals, branch_id = SFBay, 400) (Prints out error message) C_1

T1 is reading all the accounts with branch_id = SFBay and testing that the sum of balances equals the branch_total for that branch (accounts and branch_totals are in different tables)

After T1 has gone through the rows of accounts, T2 inserts another row into accounts with branch_id = SFBay (T1 will not see this as it's already scanned past the point where it is inserted), T2 then updates the branch_total for SFBay, and commits.

Now T1, having missed the new account, looks at the branch_total and sees an error.

There is no error really, just a new account row that T1 didn't see.

Note that nobody is breaking any rules about data item locks. The insert by T2 holds a write lock on a new account row that T1 never read. T2 locks the branch_total row, but then commits before T1 tries to read it.

No data item lock COULD help with this problem. But we have non-SR behavior nonetheless.

The solution is this: When T1 reads the predicate branch_id = SFBay on accounts, it takes a Read lock ON THAT PREDICATE, that is to say a Read lock on the SET of rows to be returned from that Select statement.

Now when T2 tries to Insert a new row in accounts that will change the set of rows to be returned for SFBay, it must take a Write lock on that predicate.

Clearly this Write lock and Read lock will conflict. Therefore T2 will have to wait until T1 reaches EOT and releases all locks.

So the history of Example 10.5.3 can't happen. (In reality, use a type of locking called Key-Range locking to guarantee predicate locks. Cover in Database Implementation course.)

ANSI Repeatable Read Isolation doesn't provide Predicate Locks, but ANSI Serializable does.

Note that Oracle doesn't ever perform predicate locks. Oracle's SERIALIZABLE isolation level uses a different approach, based on snapshot reads, that is beyond what we can explain in this course.

10.6 Transactional Recovery.

The idea of transactional recovery is this.

Memory is "Volatile", meaning that at unscheduled times we will lose memory contents (or become usure of the validity).

But a database transaction, in order to work on data from disk, must read it into memory buffers.

Remember that a transaction is "atomic", meaning that all update operationa a transaction performs must either ALL succeed or ALL fail.

If we read two pages into memory during a transaction and update them both, we might (because of buffering) have one of the pages go back out to disk before we commit.

What are we to do about this after a crash? An update has occurred to disk where the transaction did not commit. How do we put the old page back in place?

How do we even know what happened? That we didn't commit?

A similar problem arises if we have two pages in memory and after commit we manage to write one of the pages back to disk, but not the other.

(In fact, we always attempt to minimize disk writes from popular buffers, just as we minimize disk reads.)

How do we fix it so that the page that didn't get written out to disk gets out during recovery?

The answer is that as a transaction progresses we write notes to ourselves about what changes have been made to disk pages. We ensure that these notes get out to disk to allow us to correct any errors after a crash.

These notes are called "logs", or "log entries". The log entries contain "Before Images" and "After Images" of every update made by a Transaction.

In recovery, we can back up an update that shouldn't have gotten to disk (the transaction didn't commit) by applying a Before Image.

Similarly, we can apply After Images to correct for disk pages that should have gotten to disk (the transaction did commit) but never made it.

There is a "log buffer" in memory (quite long), and we write the log buffer out to the "log on disk" every time one of following events occur.

(1) The log buffer fills up. We write it to disk and meanwhile continue filling another log buffer with logs. This is known as "double buffering" and saves us from having to wait until the disk write completes.

(2) Some transaction commits. We write the log buffer, including all logs up to the present time, before we return from commit to the application (and the application hands out the money at the ATM). This way we're sure we won't forget what happened.

Everything else in the next few sections is details: what do the logs look like, how does recovery take place, how can we speed up recovery, etc.

10.7 Recovery in Detail: Log Formats.

Consider the following history H5 of operations as seen by the scheduler:

(10.7.1) H5 = R_1(A,50) W_1(A,20) R_2(C,100) W_2(C,50) C2 R_1(B,50) W_1(B,80) C_1

Because of buffering, some of the updates shown here might not get out to disk as of the second commit, C1. Assume the system crashes immediately after. How do we recover all these lost updates?

While the transaction was occurring, we wrote out the following logs as each operation occurred (Figure 10.13, pg. 673).

OPERA-TION	**LOG ENTRY** ***** LEAVE UP ON BOARD *****
R_1(A,50)	(S, 1) — Start transaction T_1 - no log entry is written for a Read operation, but this operation is the start of T_1
W_1(A,20)	(W, 1, A, 50, 20) — T_1 Write log for update of A.balance. The value 50 is the **Before Image** (**BI**) for A.balance column in row A, 20 is the **After Image** (**AI**) for A.balance
R_2(C,100)	(S, 2), another start transaction log entry.
W_2(C,50)	(W, 2, C, 100, 50), another Write log entry.
C2	(C, 2) — Commit T_2 log entry. (***Write Log Buffer to Log File***.)
R_1(B,50)	No log entry.
W_1(B,80)	(W, 1, B, 50, 80)
C_1	(C, 1) Commit T_1 (***Write Log Buffer to Log File***.)

Assume that a System Crash occurred immediately after the W_1(B,80) operation.

This means that the log entry (W, 1, B, 50, 80) has been placed in the log buffer, but the last point at which the log buffer was written out to disk was with the log entry (C, 2)

This is the final log entry we will find when we begin to recover from the crash. Assume that the values out on disk are A = 20 (the update to 20 drifted out to disk), B = 50 (update didn't get to disk), and C = 100 (same).

If you look carefully at the sequence, where T2 committed and T1 didn't, you will see that the values should be: A = 50, B = 50, C = 50.

After the crash, a commend is given by the system operator that initiates recovery. This is usually called the RESTART command.

The process of recovery takes place in two phases, *Roll Back* and *Roll Forward*.. The Roll Back phase backs out updates by uncommitted transactions and Roll Forward reapplies updates of committed transactions.

In Roll Back, the entries in the disk log are read backward to the beginning, System Startup, when A = 50, B = 50, and C = 100.

In Roll Back, the system makes a list of all transactions that did and did not commit. This is used to decide what gets backed out and reapplied.

LOG ENTRY	**ROLL BACK/ROLL FORWARD ACTION PERFORMED**
1.(C, 2)	Put T_2 into "Committed List"
2.(W, 2, C,100,50)	Since T_2 is on "Committed List", we do nothing.
3.(S, 2)	Make a note that T_2 is no longer "Active"
4.(W, 1, A, 50, 20)	Transaction T_1 has never committed (it's last operation was a Write). Therefore, the system performs UNDO of this update by Writing the Before Image value (50) into data item B. Put T_1 into "Uncommitted List"
5.(S, 1)	Make a note that T_1 is no longer "Active". Now that no transactions were active, we can end the ROLL BACK phase.

ROLL FORWARD

6. (S, 1)	No action required
7.(W, 1, A, 50, 20)	T_1 is Uncommitted — No action required
8.(S, 2)	No action required
9.(W, 2, C,100,50)	Since T_2 is on Committed List, we REDO this update by writing After Image value (50) into data item C
10 (C, 2)	No action required
11	We note that we have rolled forward through all log entries and terminate Recovery.

Note that at this point, A = 50, B = 50, and C = 50.

Guarantees That Needed Log Entries are on Disk

How could a problem occur with our method of writing logs and recovering? Look at the history earlier again and think what would happen if we ended up with B = 80 because the final written value of B got out to disk.

Since we have been assuming that the log (W, 1, B, 50, 80) did NOT get out to disk, we wouldn't be able to UNDO this update (which should not occur, since T1 did not commit).

This is a problem that we solve with a policy that ties the database buffer writes to the Log. The policy is called *Write-Ahead Log (WAL)*.

It guarantees that no buffer dirty page gets written back to disk before the Log that would be able to UNDO it gets written to the disk Log file.

OK, this would solve the problem of UNDOs. Do we ever have a problem with REDOs? No, because we always write the Log buffer to the Log file as part of the Commit. So we're safe in doing REDO for committed Txs.

The text has a "proof" that recovery will work, given these log formats, the RESTART procedure of Roll Back/Roll Forward, and the WAL policy.

Class 24.

10.8 Checkpoints

In the recovery process we just covered, we performed ROLLBACK to *System Startup Time*, when we assume that all data is valid.

We assume that System Startup occurs in the morning, and database processing continues during the day and everything completes at the end of day.

(This is a questionable assumption nowadays, with many companies needing to perform 24X7 processing, 24 hours a day, 7 days a week, with no time when transactions are guaranteed to not be active.)

Even if we have an 8-hour day of processing, however, we can run into real problems recovering a busy transactional system (heavy update throughput) with the approach we've outlined so far.

The problem is that it takes nearly as much processing time to RECOVER a transaction as it did to run it in the first place.

If our system is strained to the limit keeping up with updates from 9:00 AM to 5:00 PM, and the system crashes at 4:59, it will take nearly EIGHT HOURS TO RECOVER.

This is the reason for checkpointing. When a "Checkpoint" is taken at a given time (4:30 PM) this makes it possible for Recovery to limit the logs it needs to ROLLBACK and ROLLFORWARD.

A simple type of Checkpoint, a "Commit Consistent Checkpoint," merely duplicates the process of shutting down for the night, but then transactions start right up again.

The problem is that it might take minutes to take a Commit Consistent Checkpoint, and during that time NO NEW TRANSACTIONS CAN START UP.

For this reason, database systems programmers have devised two other major checkpointing schemes that reduce the "hiccup" in transaction processing that occurs while a checkpoint is being performed.

The Commit Consistent Checkpoint is improved on by using something called a "Cache Consistent Checkpoint". Then an even more complicated checkpoint called a "Fuzzy Checkpoint" improves the situation further.

So this is what we will cover now, in order (put on board):

Commit Consistent Checkpoint
Cache Consistent Checkpoint
Fuzzy Checkpoint

We define the Checkpoint Process. From time to time, a Checkpoint is triggered, probably by a time since last checkpoint system clock event.

Def 10.8.1. Commit Consistent Checkpoint steps. After the "performing checkpoint state" is entered, we have the following rules.

(1) No new transactions can start until the checkpoint is complete.

(2) Database operation processing continues until all existing transactions Commit, and all their log entries are written to disk. (Thus we are *Commit Consistent*).

(3) Then the current log buffer is written out to the log file, and after this the system ensures that all dirty pages in buffers have been written out to disk.

(4) When steps (1)-(3) have been performed, the system writes a special log entry, (CKPT), to disk, and the Checkpoint is complete. Ÿ

It should be clear that these steps are basically the same ones that would be performed to BRING THE SYSTEM DOWN for the evening.

We allow transactions in progress to finish, but don't allow new ones, and everything in volatile memory that reflects a disk state is put out to disk.

As a matter of fact, the Disk Log File can now be emptied. We needed it while we were performing the checkpoint in case we crashed in the middle, but now we don't need it any longer.

We will never need the Log File again to UNDO uncommitted transactions that have data on disk (there are no such uncommitted transactions) or REDO committed transactions that are missing updates on disk (all updates have gone out to disk already).

From this, it should be clear that we can modify the Recovery approach we have been talking about so that instead of a ROLLBACK to the Beginning of the Log File at System Startup, we ROLLBACK to the LAST CHECKPOINT!!!

If we take a Checkpoint every five minutes, we will never have to recover more than five minutes of logged updates, so recovery will be fast.

The problem is that the Checkpoint Process itself might not be very fast.

Note that we have to allow all transactions in progress to complete before we can perform successive steps. If all applications we have use very short transactions, there should be no problem.

Cache Consistent Checkpoint

But what if some transactions take more than five minutes to execute?

Then clearly we can't guarantee a Checkpoint every five minutes!!!

Worse, while the checkpoint is going on (and the last few transactions are winding up) nobody else can start any SHORT transactions to read an account balance or make a deposit!

We address this problem with something called the "Cache Consistent Checkpoint". With this scheme, transactions can continue active through the checkpoint. We don't have to wait for them all to finish and commit.

Definition 10.8.2. Cache Consistent Checkpoint procedure steps.

(1) No new transactions are permitted to start.

(2) Existing transactions are not permitted to start any new operations.

(3) The current log buffer is written out to disk, and after this the system ensures that all dirty pages in cache buffers have been written out to disk. (Thus, we are "Cache" (i.e., Buffer) Consistent on disk.)

(4) Finally, a special log entry, (CKPT, List) is written out to disk, and the Checkpoint is complete. NOTE: this (CKPT) log entry contains a list of active transactions at the time the Checkpoint occurs. Ÿ

The recovery procedure using Cache Consistent Checkpoints differs from Commit Consistent Checkpoint recovery in a number of ways.

Ex 10.8.1 Cache Consistent Checkpoint Recovery.

Consider the history H5:

H5: R_1(A, 10) W_1(A, 1) C_1 R_2(A, 1) R_3(B, 2) W_2(A, 3) R_4(C, 5) CKPT
W_3(B, 4) C_3 R_4(B, 4) W_4(C, 6) C_4 CRASH

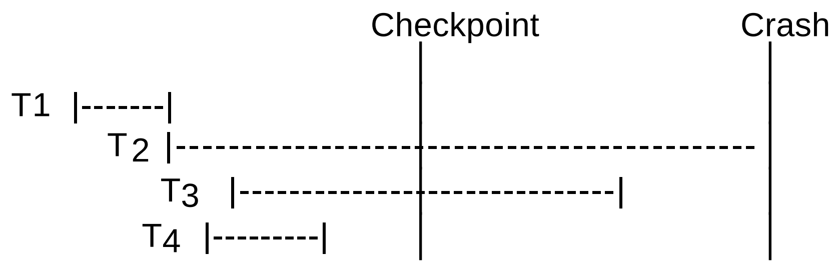
Here is the series of log entry events resulting from this history. The last one that gets out to disk is the (C, 3) log entry.

(S, 1) (W, 1, A, 10, 1) (C, 1) (S 2) (S, 3) (W, 2, A, 1, 3) (S, 4)
(CKPT, (LIST = T_2, T_4)) (W, 3, B, 2, 4) (C, 3) (W, 4, C, 5, 6) (C, 4)

At the time we take the Cache Consistent Checkpoint, we will have values out on disk: A = 3, B = 2 , C = 5. (The dirty page in cache containing A at checkpoint time is written to disk.)

Assume that no other updates make it out to disk before the crash, and so the data item values remain the same.

Here is a diagram of the time scale of the various events. Transaction T_k begins with the (S, k) log, and ends with (C, k). **LEAVE ON BOARD**



Next, we outline the actions taken in recovery, starting with ROLL BACK.

ROLL BACK

1.(C, 3) Note T_3 is a committed Tx in active list.
2.(W, 3, B, 2, 4) Committed transaction, wait for ROLL FORWARD.
3.(CKPT, (LIST = T_2, T_4)) Note active transactions T_2 and T_4;
THESE HAVE NOT COMMITTED (no (C, 2) or
(C, 4) logs have been encountered)
4.(S, 4) List of Active transactions now shorter: {T_2, T_3}
5.(W, 2, A, 1, 3) Not Committed. UNDO: A = 1
6.(S, 3) List of Active Transactions shorter.
7.(S, 2) List of Active Transactions empty. STOP ROLLBACK.

With a Cache Consistent Checkpoint, when ROLL BACK encounters the CKPT log entry the system takes note of the transactions that were active, even though we have never seen any operations in the log file.

We now take our list of active transactions, remove those that we have seen committed, and have a list of transactions whose updates we need to UNDO. Since Transactions can live through Checkpoints we have to go back PRIOR to the Checkpoint, while UNDO steps might remain.

We continue in the ROLL BACK phase until we complete all such UNDO actions. We can be sure when this happens because as we encounter (S, k) logs, rolling backward.

When all Active Uncommitted T_k have been removed, the ROLL BACK is complete, even though there may be more entries occurring earlier in the log file.

ROLL FORWARD

8.(CKPT, (LIST = T_2, T_3)) Skip forward in log file to this entry, start after this.
9.(W, 4, C, 5, 6) Roll Forward: C = 6.
18.(C, 4) No Action. Last entry: ROLL FORWARD is complete.

In starting the Roll Forward Phase, we merely need to REDO all updates by committed transactions that might not have gone out to disk.

We can jump forward to the first operation after the Checkpoint, since we know that all earlier updates were flushed from buffers.

Roll Forward continues to the end of the Buffer File. Recall that the values on disk at the time of the crash were: A = 3, B = 2 , C = 5. At the end of Recovery, we have set A = 1 (Step 5), and C = 6 (Step 9).

We still have B = 4. A glance at the time scale figure shows us that we want updates performed by T_4 to be applied, and those by T_2 and T_3 to be backed out. There were no writes performed by T_4 that got out to disk, so we have achieved what is necessary for recovery: A = 1, B = 4, C = 6.

Fuzzy Checkpoint.

A problem can still arise that makes the Cache Consistent Checkpoint a major hiccup in Transaction Processing.

Note in the procedure that we can't let any Active Transactions continue, or start any new ones, until all buffers are written to disk. What if there are a LOT of Buffers?

Some new machines have several GBytes of memory! That's probably Minutes of I/O, even if we have a lot of disks. DISK I/O is SLOW!!!

OK, with a Fuzzy Checkpoint, each checkpoint, when it completes, makes the PREVIOUS checkpoint a valid place to stop ROLLBACK.

Definition 10.8.3. Fuzzy Checkpoint procedure steps.

(1) Prior to Checkpoint start, the remaining pages that were dirty at the prior checkpoint will be forced out to disk (but the rate of writes should leave I/O capacity to support current transactions in progress; there is no critical hurry in doing this).

(2) No new transactions are permitted to start. Existing transactions are not permitted to start any new operations.

(3) The current log buffer is written out to disk with an appended log entry, ($CKPT_N$, List), as in the Cache Consistent Checkpoint procedure.

(4) The set of pages in buffer that have become dirty since the last checkpoint log, $CKPT_{N-1}$, is noted.

This will probably be accomplished by special flags on the Buffer directory. There is no need for this information to be made disk resident, since it will be used only to perform the next checkpoint, not in case of recovery. At this point the Checkpoint is complete. Ÿ

As explained above, the recovery procedure with Fuzzy Checkpoints differs from the procedure with Commit Consistent Checkpoints only that ROLL FORWARD must start with the first log entry following the SECOND to last checkpoint log. We have homeowork on this.

Class 25.

Covered last time the various Checkpoints: Commit Consistent, Cache Consistent, and Fuzzy. Any questions?

What really happens with commercial databases. Used to be all Commit Consistent, now often Fuzzy.

Also used to be VERY physical. BI and AI meant physical copies of the entire PAGE. Still need to do this sometimes, but for long-term log can be more "logical".

Instead of "Here is the way the page looked after this update/before this update", have: this update was ADD 10 to Column A of row with RID 12345 with version number 1121.

Version number is important to keep updates idempotent.

Note that recovery is intertwined with the type of recovery. It doesn't do any good to have row-level locking if have page level recovery.

T1 changes Row 12345 column A from 123 to 124, and log gives PAGE BI with A = 123, T1 hasn't committed yet.

T2 changes Row 12347 (same page) column B from 321 to 333 and Commits, log gives Row 12345 with 124, Row 12347 column B with 333 on page AI.

Transaction T2 commits, T1 doesn't, then have crash. What do we do?

Put AI of T2 in place? Gives wrong value to A. Put BI of T1 in place? Wrong value of B.

Page level logging implies page level locking is all we can do.

Sybase SQL Server STILL doesn't have row-level locking.

10.9 Media Recovery

Problem is that Disk can fail. (Not just stop running: head can score disk surface.) How do we recover?

First, we write our Log to TWO disk backups. Try to make sure two disks have Independent Failure Modes (not same controller, same power supply).

We say that storage that has two duplicates is called *stable storage*, as compared to *nonvolatile storage* for a normal disk copy.

Before System Startup run BACKUP (bulk copy of disks/databases).

Then, when perform Recovery from Media failure, put backup disk in place and run ROLL FORWARD from that disk, as if from startup in the morning.

As if normal recovery on this disk except that all pages on this disk were VERY popular and never got out from disk. Don't need Roll Back except conceptually.

RAID Disks

Something they have nowadays is RAID disks. RAID stands for Redundant Arrays of Inexpensive Disks. Invented at Berkeley. The Inexpensive part rather got lost when this idea went commercial.

The simplest kind, RAID 1, mirrors Writes. Could have two disks that write everything to in two copies. Every time we Write, need 2 Writes.

So if one disk lost, just use other disk. Put another blank disk in for mirror, and while operating with normal Reads and Writes do BACKUP to new disk until have mirror.

This approach saves the time needed to do media recovery. And of course works for OS files, where there IS no media recovery.

You can buy these now. As complex systems get more and more disks, will eventually need RAID. The more units there are on a system, the more frequent the failures.

Note that mirrored Writes are handled by controller. Doesn't waste time of System to do 2 Writes.

But when Read, can Read EITHER COPY. Use disk arms independently.

So we take twice as much media, and if all we do is Writes, need twice as many disk arms to do the same work.

But if all we do is Reads, get independent disk arm movements, so get twice as many Reads too.

But in order for twice as many Reads to work, need warm data, where disk capacilty is not the bottleneck, but disk arm movement is.

Definitely lose the capacity in RAID 1. But if we were only going to use half the capacity of the disk because we have too Many Reads, RAID 1 is fine.

There is an alternative form of RAID, RAID 5, that uses less capacity than mirroring. Trick is to have 6 disks, 5 real copies of page, one checksum.

Use XOR for Checksum. CK = D1 XOR D2 XOR D3 XOR D4 XOR D5.

If (say) D1 disappears, can figure out what it was:

D1 = CK XOR D2 XOR D3 XOR D4 XOR D5

(Prove this: A = B XOR C, then B = A XOR C and C = A XOR B.
1 = 1 XOR 0 => 1 = 1 XOR 0 and 0 = 1 XOR 1
1 = 0 XOR 1 => 0 = 1 XOR 1 and 1 = 1 XOR 0
0 = 1 XOR 1 => etc.
0 = 0 XOR 0)

So if one disk drops out, keep accessing data on it using XOR of other 5. Recover all disk pages on disk in same way. This takes a lot of time to recover, but it DOES save disk media.

10.10 TPC-A Benchmark

The TPC-A Benchmark is now out of date. Newer TPC-C Benchmark: more complex.

See Fig 10.16, pg. 686. Size of tables determined by TPS.

See Fig 10.17. pg. 687. All threads do the same thing. Run into each other in concurrency control because of Branch table and History table.

Benchmark specifies how many threads there are, how often each thread runs a Tx, costs of terminals, etc.

On a good system, just add disks until use 95% of CPU. On a bad system, run into bottlenecks.

Ultimate measure is TPS and $/TPS