Lecture 7: Concurrency Control and Recovery: based on transactions

Consistency and Isolation: Concurrency Control

- Serial Schedules
- The goal: Serializability/Isolation
- The theory:
  - Conflict Serializability
  - Precedence graphs
- The real world:
  - Locking
  - Strict Two Phase Locking
  - Deadlock
  - Isolation levels

Atomicity & Durability: Crash Recovery

- Crash Recovery: Problems
- The solution: Write Ahead Logging (WAL)
  - Atomicity
  - Durability
- ARIES
- Using the WAL to manage aborts

Transaction

- Both concurrency control and crash recovery are based on the concept of a transaction.
- A transaction is a set of SQL statements chosen by the user.
- You don’t need concurrency control and you don’t need (sophisticated) recovery if your database is read-only.

  Concurrency control and recovery are intended for databases that are updated.

Example Transaction

Transfer $100 from one account to another:

- **BEGIN** transaction
- read balance from first account
- add $100 to first account balance
- write balance to first account

using first account

- read balance in second account
- verify balance to see if it contains at least $100
  - if not, **ABORT** transaction
- subtract $100 from second account
- write balance to second account

using second account

- **COMMIT** transaction

Transaction (cont.)

- User (application developer) must indicate:
  - Begin transaction
  - read/write/modify statements intermixed with other programming language statements
- plus either
  - commit - indicates successful completion or
  - abort - indicates program wants to roll back (erase the previous steps of the transaction)
- In order to ensure the correctness of the database, the DBMS and the programmer must guarantee four properties of transactions, called the ACID properties.

Learning objectives

- Define the ACID properties and give an example of how each one could be violated.
- Recognize serial, serializable and conflict serializable schedules
- Given a write ahead log, describe what actions would ensure atomicity and durability
- Use a write ahead log to manage an abort

The ACID Properties of Transactions

- Atomicity: A transaction happens in its entirety or not at all.
- Consistency: If the DB starts in a consistent state, (this notion is defined by the user; some of it may be enforced by integrity constraints) the transaction will transform it into a consistent state.
- Isolation: Each transaction is isolated from other transactions. The effect on the DB is as if the transaction executed by itself.
- Durability: If a transaction commits, its changes to the database state persist (changes are permanent)

I and C (of ACID) Properties (cont.):

- Isolation: Each transaction is isolated from other trans. The effect on the DB is as if the transaction executed by itself.
  - What if another transaction computed the total bank balance after $100 was deposited to the first account?
  - The concurrency control subsystem must ensure isolation, unless the DBA chooses a less strict level of isolation.
- Consistency: If the DB starts in a consistent state, (this notion is defined by the user; some of it may be enforced by integrity constraints) the transaction will transform it into a consistent state.
  - What if a transaction just deposited $100 into an account without subtracting $100 from the other account?
  - The *programmer* must ensure that a transaction is consistent.

A and D (of ACID) Properties: Recovery Manager

- Atomicity: A transaction happens in its entirety or not at all
  - What if the OS crashed after $100 was deposited to the first account?
  - The *recovery manager* of the DBMS must assure that the $100 is withdrawn from the first account.
- Durability: If a transaction commits, its changes to the database state persist (changes are permanent)
  - What if, after the commit, the OS crashed before the withdrawal was written to disk?
  - The recovery manager must assure that the withdrawal was at least logged.

Concurrency

- First we will study isolation of transactions, ensured by the concurrency control subsystem.
  - For a while, we will assume that there are no updates to the database, that it is read-only.
- Isolation is a problem only when multiple users are accessing the same data, and their actions interleave.
- Why is concurrency necessary?
  - Applications demand it
  - Better utilization of resources: While one user/transaction is reading the disk, another can be using the CPU or reading another disk. This results in better throughput and response time.

Serial Schedules

- Consider these three transactions

T1: BEGIN A+=100, B-=100 END //Transfer $100
T2: BEGIN C = A+B END //Compute bank balance
T3: BEGIN A =1.06*A, B=1.06*B //Give interest

- A schedule of transactions is an interleaving of the actions of the transactions so that each transaction's order is preserved. (If order is not preserved, it's not a schedule.)
- A schedule of transactions is serial if its transactions occur consecutively, one after another.

Schedule, Serial Schedules

- Which of these is a schedule? A serial schedule?



T1: A+=100, B-=100	
T3:	A=1.06*A, B=1.06*B

Schedule for T1 and T3. Serial.

T1: A+=100,		B-=100
T3:	A=1.06*A,	B=1.06*B

Schedule for T1 and T3. Not serial.

T1: B-=100, A+=100	
T2:	C=A+B

Not a schedule because T1 actions are out of order.

Which (non-serial) schedule is correct (from an isolation point of view)?



S1

T1: A+=100, B-=100
T3: A=1.06*A, B=1.06*B

(Correct) serial schedule for T1 & T3

S2

T1: A+=100, B-=100
T3: A=1.06*A, B=1.06*B

Non-serial schedule for T1, T3

S3

T1: A+=100, B-=100
T2: C=A+B

(Correct) serial schedule for T1 & T2.

S4

T1: A+=100, B-=100
T2: C=A+B

Non-serial schedule for T1 & T2.

Allowable Concurrency

- Is S2 correct? (What do we mean by correct?)
- Is S4 correct?
- We want to allow interleaved schedules like S2, otherwise the DBMS is serial/slow!
- **How can we distinguish between S2 and S4?**
- The DBMS should certainly allow serial schedules like S2 and forbid interleaved schedules like S4.

Serializable Schedules

- A schedule is serializable (correct) if its effect on the DB is the same as the effect of some serial schedule.
  - Serial schedules are always serializable
  - S2 is serializable (but S4 is not)
- Serializability provides the isolation condition of ACID.
- The goal of the concurrency control subsystem of a DBMS is to ensure serializability*
- First we will study the theory of serializability, then we will see how it is ensured in practice.

*In some circumstances, conditions weaker than serializability are appropriate.

Schedules as reads and writes

- The expression A =1.06*A in the previous schedule means
  - Read A from disk
  - Set A equal to 1.06*A
  - Write A to disk
- Only the read and write to disk matter to the DBMS.
  - We use the notation R(A), W(A)
- So henceforth we write schedules in terms of reads and writes to the DBMS (and leave out arithmetic or other processing)
- This captures the DBMS activity more directly

Schedules as reads & writes (ctd.)

- Recall S2:

S2

T1: A+=100,		B-=100
T3:	A=1.06*A,	B=1.06*B

- This can be rewritten as:

T1: R(A),W(A),		R(B),W(B)
T3:	R(A),W(A),	R(B),W(B)

Conflict serializability (a method to prove serializability)

Consider a schedule.

- Two actions are nonconflicting if they are in different transactions and either they access different data items or both are reads. (The green actions are nonconflicting on prior slide.)

- If nonconflicting actions are commuted/swapped then new schedule gives the same result as the old one.

- Two schedules are **conflict equivalent** if one can be transformed into the other by commuting nonconflicting actions
- A schedule is **conflict serializable** if it is conflict equivalent to a serializable schedule
- Therefore every conflict serializable schedule is serializable

Proving Serializability

- Recall S2:

S2

```
T1: R(A),W(A),                R(B),W(B)
T3:             R(A),W(A),                R(B),W(B)
```

- Why is S2 serializable?
  Since the R(A),W(A) in T3 is using data item A and the R(B),W(B) in T1 is using data item B, there is NO CONFLICT between these actions. So we can commute them; i.e., move R(B) and W(B) earlier.

```
T1: R(A),W(A), R(B),W(B)
T3:                       R(A),W(A),  R(B),W(B)
```

This is a serial schedule! So the original schedule is "conflict serializable" and thus is serializable.

Proving Serializability (cont.)

- But S4 is not serializable.

S4

T1: A+=100, B-=100
T2: C=A+B

S4 rewritten

T1: R(A),W(A) R(B),W(B)
T2: R(A),R(B)

S4 after swap

T1: R(A),W(A) R(B), W(B)
T2: R(A), R(B)

- We can't move R(A) to the left (it's conflicting)
- We can move R(B) one spot to the right, but not two.
- We can't swap the schedule until it's serial.

Precedence graphs

- Verifying conflict serializability is laborious. But there is an easier way.
- Precedence graph:
  Create one node per transaction. Create one edge from Ti to Tj if an action in Ti occurs before an action in Tj and they conflict.
- Theorem: A schedule is conflict serializable if and only if its precedence graph is **acyclic (not cyclic)**.
- Draw the precedence graph for the previous schedules.

Is this schedule serializable?
Draw a precedence graph; check for cycles.

S2

T1: R(A),W(A),			R(B),W(B)
T2:	R(A),W(A),		R(B),W(B)



First set of conflicting actions:
T1 reads/writes A; T2 reads/writes A.
Since T1's action occurs before T2 in the schedule, draw an arrow from T1 to T2.

Second set of conflicting actions: T1 reads/writes B before T2; we already have an arrow from T1 to T2.
We're finished drawing the graph.
There are no cycles! Therefore, schedule is serializable.

Exercise: check to see if this schedule is serializable; draw a precedence graph.

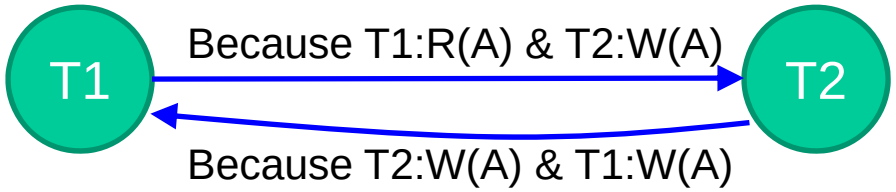
S4

T1: R(A),W(A)		R(B),W(B)
T2:	R(A),R(B)	

LO7.2: Which of these schedules is serializable?*

S5

T1:	R(A),		W(A)
T2:	R(A),	W(A), R(B)	



There is a cycle!
Not serializable.

S6

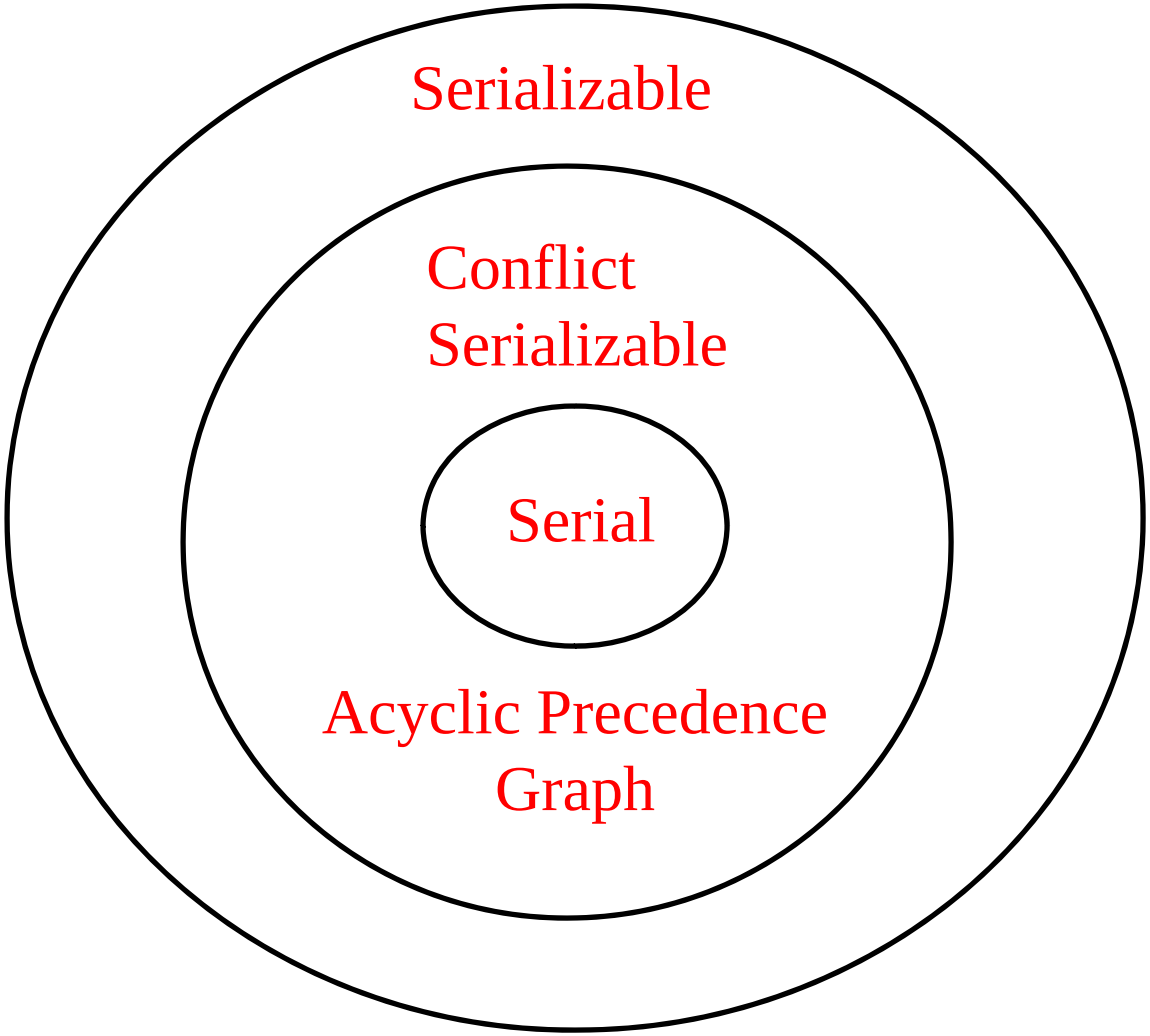
T1:	R(A),	R(B)	W(A)
T2:	W(A)		
T3:		W(A)	R(B)

S7

T1:	R(A),	W(A)
T2:	W(A)	
T3:		W(A)

Check for serializability

Conclusions



Serializability in the real world

- Now we have a simple way to prove that a schedule is serializable: Precedence graphs. It does not work for all serializable schedules (see S9) but it works for most of them. This helps us understand serializability. In theory, a DBMS could use this method to check every schedule for serializability.
- But a real DBMS is not presented with schedules, it sees only a stream of transactions. What can a real DBMS do to enforce serializability and thus achieve the “isolation” ACID property?

Locking: Used by real DBMSs to enforce serializability

- Transaction must get a lock – before it can read or update data
- There are two kinds of locks:
  shared (S) locks and exclusive (X) locks
- To read a record you MUST get an S lock
  To write (modify or delete) a record you MUST get an X lock
- Lock info maintained by a “lock manager”

How Locks Work

- If an object has an S lock, new transactions can get S locks but not X locks.
- If an object has an X lock, no other transaction can get any lock (S or X) on that object.
- If a transaction can't get a lock, it *waits* (in a queue).

lock on data item

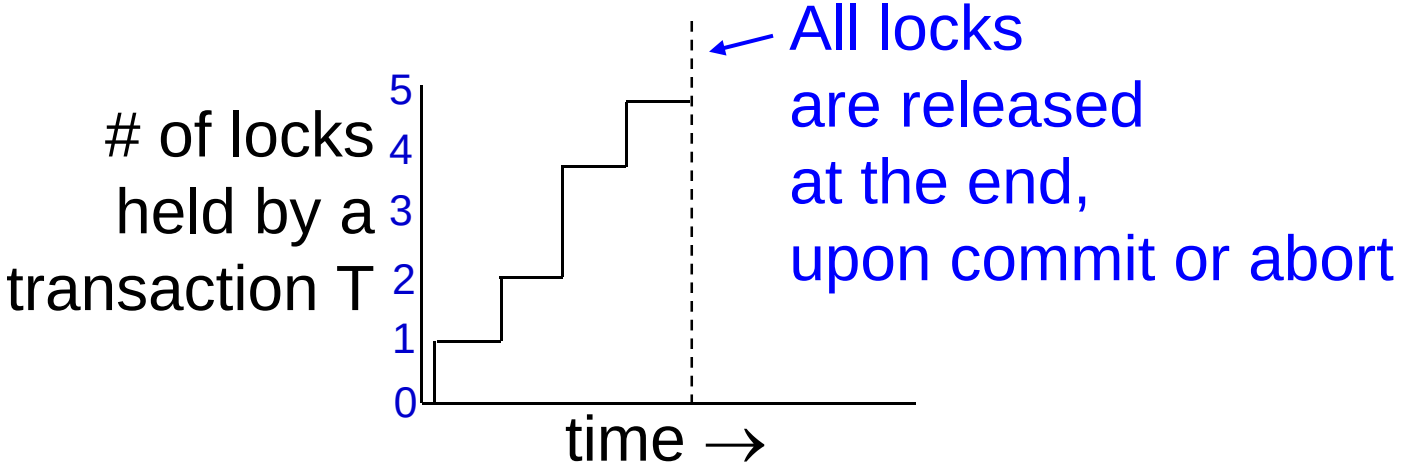
lock you want	--	S	X
--	ok	ok	ok
S	ok	ok	no
X	ok	no	no

Lock compatibility

Strict Two Phase Locking Protocol (S2PL)

Strict 2PL is a way of managing locks during a transaction

- T gets (S and X) locks gradually, as needed
- T hold all locks until end of transaction (commit/abort)



Strict 2PL guarantees serializability

- Proof: a Strict 2PL schedule is equivalent to the serial schedule in which each transaction runs instantaneously at the time that it commits
- This is huge: A property of each transaction (S2PL) implies a property of any set of transactions (serializability)
  - No need to check serializability of any schedules
- Real DBMSs use S2PL to enforce serializability

Deadlock in DBMSs

- What is a deadlock?
  - A *cycle* of transactions, T_1, T_2, ... , T_n=T_1 where each T_i is *waiting* for T_{i-1} to release a lock.
  - Causes these transactions to sleep/wait forever.
- A Deadlock can happen whenever you allow a transaction to wait for a lock, even with strict two phase locking.
  - Simple example:

T1:	R(B),	W(A)
T2: R(A),		W(B)

- Users can eliminate deadlocks by accessing resources in a fixed order.
- DBMSs typically detect deadlocks and abort the transaction that (it thinks) has used the least resources.

Isolation levels

Developers can choose how much isolation (protection) they want … There are four isolation levels defined in the SQL standard. They involve concepts that we will not cover in this course:

- "READ UNCOMMITTED" – allows dirty read, unrepeatable read, and "phantoms"
- "READ COMMITTED*" – allows unrepeatable reads and phantoms
- "REPEATABLE READ" – allows phantoms
- "SERIALIZABLE*" – full isolation

*These are the only levels available in Postgres

The Recovery System: (Review the ACID properties)

Recovery System

Programmers

Concurrency Control System

Recovery System

Atomicity: The actions in a transaction must ALL happen or NONE of them happen.

Consistency: If each transaction is consistent, and the DB starts in a consistent state, it ends in a consistent state.

Isolation: Execution of one transaction is isolated from that of other transactions.

Durability: If a transaction commits, its effects persist.

Recovery Basics

- Updates are happening "in place." That is, data is overwritten on (or deleted from) the DB disk directly.
- The problem is … when you INSERT, DELETE, or UPDATE a record in the DB; you are actually modifying data in a buffer (that holds a copy of the DB page). When will that buffer be copied back out to the DB?
- We don't know!
- If the system crashes, none, some, or all of the updates from a transaction may be in the DB. Thus, we can violate atomicity and durability.

The Crash Recovery Manager's Problems

- Recall the typical transaction
  - $100 deposited to A, then $100 withdrawn from B
- Recall the Recovery Manager's problems:
  - OS crashes after the deposit
    - If deposit was written to the DB disk (but withdrawal wasn't)
    - Bank loses $100
    - Violates Atomicity
  - OS crashes after commit
    - If neither deposit nor withdrawal was written to the DB disk, transaction appears to have committed (you have your receipt) but nothing actually happened in the DB.
    - Violates Durability
- Any ideas about how to solve these problems?

The solution

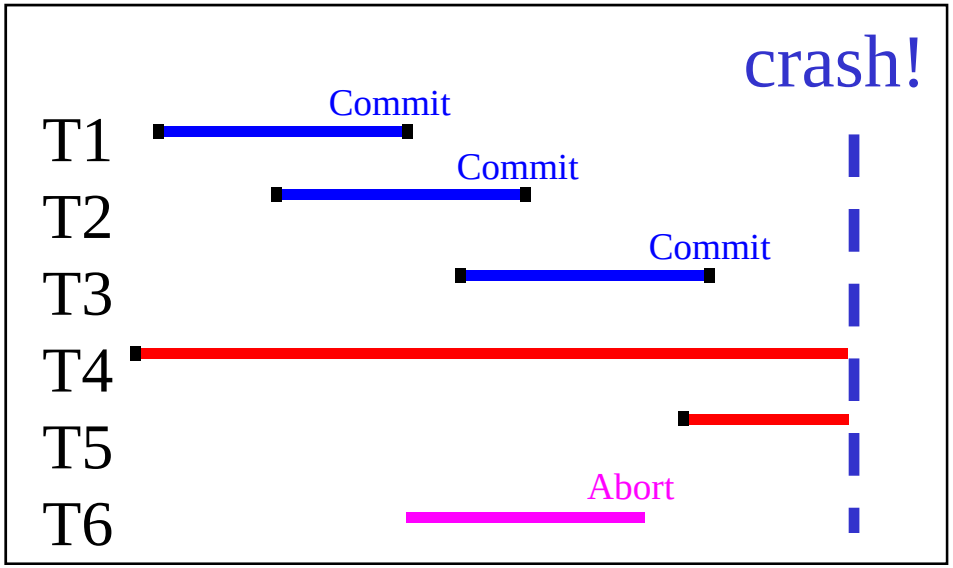
- The crash recovery subsystem of every DBMSs uses a Write Ahead Log (WAL) to manage crash recovery (and aborts also).
- The WAL is put on a separate disk from the data (why?). It begins after each backup, which might be taken each night.
- A log record is written for every insert, update, delete, begin-trans, commit, abort and checkpoint.
- A log record contains

<XID, ID of the DB record, action, old data, new data>

before image

after image

Motivation for Recovery: What the WAL should help with



- Desired Behavior when system restarts after this crash:
  - T1, T2 & T3 committed (before the crash); they must be durable. (WAL should REDO these transactions)
  - T4 & T5 were still running and must be aborted (effects must not be seen). (WAL should UNDO these transactions)
  - T6 was aborted before the crash. (WAL should UNDO T6)

Write Ahead Log (WAL)

To be a write ahead log, the log must obey these rules

The Atomic Rule: The log entry for an insert, update or delete must be written to (log) disk before the change is made to the DB

The Durability Rule: All log entries for a transaction must be written to (log) disk before the commit record is written to (log) disk.

Practice with a log

```
T1,A,update,100,200
T2,C,update,1000,500
T2,D,update,500,1000
T2,commit
CRASH
```

- What did each transaction do before the crash?
- After the crash, what should the recovery manager do to ensure that each transaction is atomic?
  -
  - Which WAL rule guarantees that your solution (UNDO)will work?
- After the crash, what should the recovery manager do to ensure that each transaction is durable?
  -
  - Which WAL rule guarantees that your solution (REDO) will work?

Answers to previous slide

- What did each transaction do before the crash?
  - T1 added 100 to A.
  - T2 subtracted 500 from C. Then T2 added 500 to D. Then T2 committed.
- After the crash, how does the recovery manager make sure that each transaction is atomic (all or nothing)?
  - It should UNDO (roll back) each transaction that has not committed.
  - For a, it will read the log and will then update (the DB copy of) A to 100. We don't know if A is 100 or 200 but we KNOW that it should be 100 now.
  - The atomic rule guarantees that the log record was written before the DB.
- After the crash, what should the recovery manager do to ensure that each transaction is durable?
  - It should REDO the actions of all committed transactions: update (the DB copy of) C to 500 and update (the DB copy of) D to 1000.
  - The durability rule guarantees that all log records for the entire transaction are in the log; they were written before the commit record was written to the log.

LO7.3: Use a log

- What must a recovery manager do after a crash to ensure the atomic and durability properties of ACID?
- What are the final values of A and D?
- Does the recovery manager return the DB to its state at the time of the crash?
- Not necessarily; some of T1 updates may have made it to the DB disk.

```
T1,A,update,'abc','de'
T1,A,update,'de','ab'
T2,D,update,10,0
T2,commit
CRASH
```

Real recovery is more complicated

- We have ignored many complexities in crash recovery
  - Managing normal aborts, some of which may be in progress at the time of the crash
  - Managing inserts and deletes
  - Supporting multiple lock levels
  - Managing updates to structures like B+ trees when pages split
  - Handling crashes that happen in the middle of recovery
- In the early days of DBMSs many inflexible, inefficient and even incorrect recovery algorithms were implemented.
- But mechanisms for correct recovery were invented and implemented.

ARIES

- In the early 1990s, C. Mohan of IBM proposed a recovery algorithm called ARIES*
- ARIES differs from our simple model in a few ways
  - It redoes every update, not just those of committed transactions. Surprisingly, this simplifies the algorithm.
  - It logs changes when normal aborts are undone and it logs undos during recovery. This speeds up a second (or third or ..) recovery if ever you crash during the earlier recoveries.
  - And more…see CS410/587

* Algorithms for Recovery and Isolation Exploiting Semantics

WAL is also used for regular aborts

- We have seen that a Write Ahead Log makes atomicity and durability easy to achieve.
- A Write Ahead Log also makes transaction abort simple.
- A transaction does not have to keep track of the changes it has made to the DB so it can undo them in case of abort.
- The DBMS just uses the Write Ahead Log and does a regular undo! (The DBMS applies the “before images” from the WAL in reverse order, for that transaction.)

Aborting a transaction

```
T1,A,update,'ABC','DEF'
T2,C,update,1000,500
T2,D,update,500,1000
T1,B,update,300,400
T1,A,update,'DEF','GHI'
T1,abort
```

- Note that this is normal processing – no crash in sight
- What actions must the DBMS take to abort T1?
- In what order should these actions be taken?
- What guarantees that all of T1's changes to the DB have been undone?
- What if the update to B was not written to the DB disk yet?

Database Backup

- As with any other file, periodically backup your database.
  - Backup – complete copy of your database.
  - Keep more than one copy; keep at least one copy off-site.
  - Simple case (but now always possible) – make a backup copy while your database is inactive (i.e., in a quiesce point).
- Database logs plus database backup allow recovery from catastrophic failure – when you lose your database.
  - Restore the database from the most recent backup.
  - Apply all the logs created since the backup, in timestamp order, doing a REDO operation on all transactions that committed – in forward order. (Ignore aborted trans.)
  - Keep a complete set of logs – to allow this kind of recovery.

What is a Checkpoint

- Periodically, the DBMS creates a checkpoint,.
  - (simple case) flush all DB pages to disk - so that we know all DB updates described in log records prior to the checkpoint have been written to disk
  - (simple case) wait for quiesce point: all transactions finished. (Aries can create a checkpoint without requiring a quiese point.)
  - write checkpoint record to the log
- Checkpoint record = DB is consistent at this point! Checkpoint minimizes the time taken to recover from a crash - because it helps us know how far we need to go backwards in the log.

What if there is a crash during recovery?

- During the recovery, the system might crash
- Can we start the recovery process over again, from the beginning? Will it still work correctly?
- Restating the same question, does anything unexpected happen if we do part of the recovery work twice (or more times)?
- Restating in the form of a requirement: We need the recovery process to be idempotent so that no matter how many times we run it, it provides the same (correct) answer.

Exercise #1: Which of these schedules is serial, serializable, or conflict serializable?

S10

```
T1: R(C),W(C)
T2:                          R(C),W(C), W(A)
T3:              W(C), R(A)
```

S11

```
T1:          R(B),W(C)                                  R(E),W(E)
T2:                   R(B),R(C),          R(D),W(D)
T3:R(A),W(B)
T4                                W(C),W(B)                        R(E),W(E)
```

S12

```
T1:          R(B),W(C)                                  R(E),W(E)
T2:                   R(B),R(C),          R(D),W(D)
T3:R(A),W(B),                                                     R(D)
T4                                W(C),W(B)                        R(E),W(E)
```

Exercise #2: Aborting a transaction*

```
T1,A,update,100,200
T1,A,update,200,100
T2,D,update,500,1000
T1,B,update,'ABC','DEF'
T1,abort
```

- What actions must the DBMS take, in what order, to abort T1?