Transaction Management

Ramakrishnan & Gehrke, Chapter 14+

Transactions

- Concurrent execution of user requests is essential for performance
  - User requests arrive concurrently
  - disk accesses frequent + slow: important to keep CPU humming by working on several application programs concurrently
- Application program may carry out many operations on data retrieved, but DBMS only concerned about data read/written from/to database
- transaction (TA) = DBMS's abstract view of user program: sequence of (SQL) reads & writes executed as a unit

Concurrency in a DBMS

- Users submit TAs, can think of each (trans)action as execution unit
  - Concurrency achieved by DBMS by interleaving TAs
  - TA must leave DB in consistent state
    assuming DB is consistent when TA begins
    - *ICs declared in CREATE TABLE, CHECK constraints, etc.*
- Issues:
  - Effect of interleaving TAs
  - Crashes
  - Performance of concurrency control

Atomicity of Transactions

- Two possible TA endings:
  - commit after completing all its actions – data must be safe in DB
  - abort (by application or DBMS) – must restore original state
- Important property guaranteed by the DBMS: TAs atomic
  - Perception: TA executes all its actions in one step, or none
- Technically: DBMS logs all actions
  - can undo actions of aborted TAs

ACID

- TA concept includes four basic properties:
- Atomic
  - all TA actions will be completed, or nothing
- Consistent
  - after commit/abort, data satisfy all integrity constraints
- Isolation
  - any changes are invisible to other TAs until commit
- Durable
  - nothing lost in future; failures occurring after commit cause no loss of data

Transaction Syntax in SQL

- START TRANSACTION start TA
- COMMIT end TA successfully
- ROLLBACK abort TA (undo any changes)
- If none of these TA management commands is present, each statement starts and ends its own TA
  - including all triggers, constraints,…

Anatomy of Conflicts

- Consider two TAs:

```
T1:     BEGIN  A=A-100,   B=B+100   END
T2:     BEGIN  A=1.06*A,  B=1.06*B  END
```

  - Intuitively, first TA transfers $100 from B's account to A's account
  - second TA credits both accounts with a 6% interest payment

- no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together

- However, net effect must be equivalent to these two TAs running serially in some order

Anatomy of Conflicts (contd.)

- Consider a possible interleaving (schedule):

T1:	A=A-100,		B=B+100	
T2:		A=1.06*A,		B=1.06*B

- This is OK. But what about:

T1:	A=A-100,		B=B+100
T2:		A=1.06*A, B=1.06*B	

- The DBMS's view of the second schedule:

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

Anomalies from Interleaved Execution

- Reading uncommitted data (R/W conflicts, "dirty reads"):

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

- Unrepeatable reads (R/W conflicts):

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

- Overwriting uncommitted data (W/W conflicts):

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

Scheduling Transactions: Definitions

- Serial schedule:
  Schedule that does not interleave the actions of different TAs
- Equivalent schedules:
  For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule
- Serializable schedule:
  A schedule equivalent to some serial execution of the TAs
- each TA preserves consistency
  $\Rightarrow$ every serializable schedule preserves consistency

Lock-Based Concurrency Control

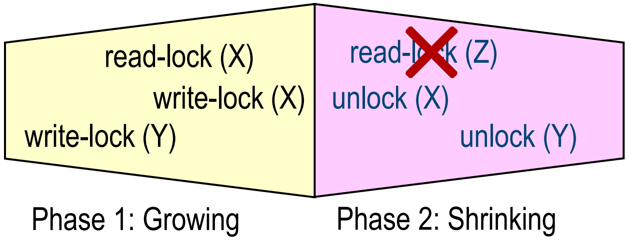
- Core issues: What lock modes? What lock conflict handling policy?
- Common lock modes: SX
  - Each TA must obtain an S (shared) lock before reading, and an X (exclusive) lock before writing

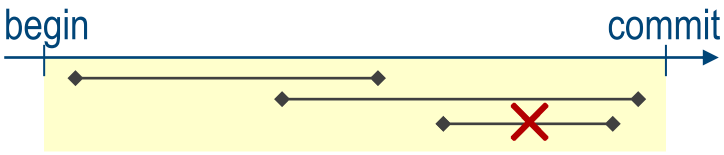
```
   | S X
 --+------
 S | + -
 X | - -
```

- Lock conflict handling
  - Abort conflicting TA / let it wait / work on previous version
- Locking protocols
  - two-phase locking (strict, non-strict, conservative, …) – *next!*
  - Timestamp based
  - Multi-version based
  - Optimistic concurrency control

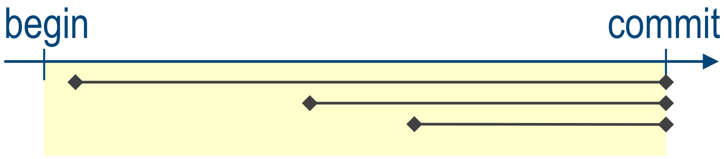
Two-Phase Locking Protocol

- 2PL
  - All locks acquired before first release
  - cannot acquire locks after releasing first lock
- allows only serializable schedules ☺
  - but complex abort processing



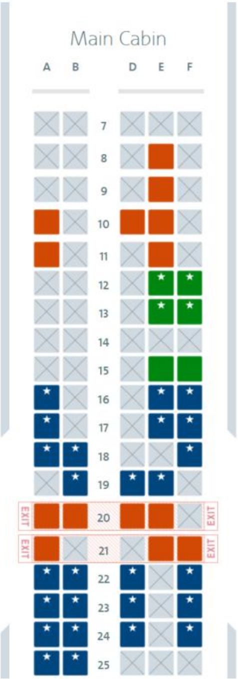


- Strict 2PL
  - All locks released when TA completes
- Strict 2PL simplifies TA aborts ☺☺



Isolation Levels

- Isolation level directives: summary about TA's intentions, placed before TA
  - SET TRANSACTION READ ONLY
    TA will not write → can be interleaved with other read-only TAs
  - SET TRANSACTION READ WRITE
    (default)
- assists DBMS optimizer
- Example: Choosing seats in airplane
  - *Find free seat, reserve by occ:=TRUE; if there is none, abort*
  - *customer approval → commit, otherwise release seat by occ:=FALSE, try again*
  - two "TA"s concurrently: can have dirty reads for occ – uncritical! (why?)

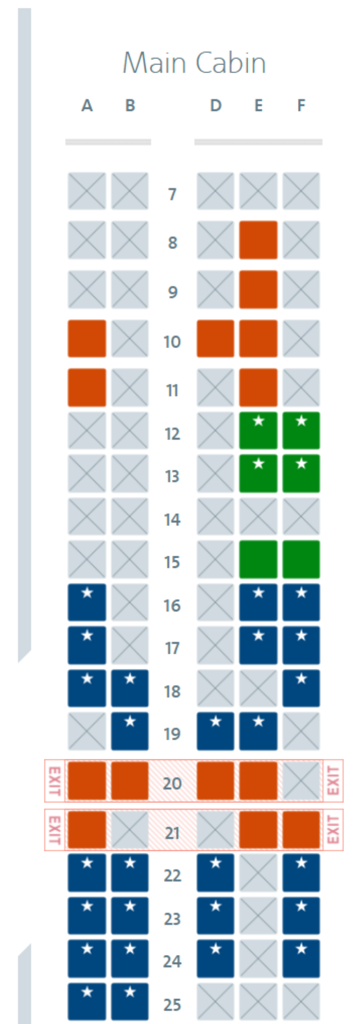


Isolation Levels (contd.)

- Refinement:
  SET TRANSACTION READ WRITE ISOLATION LEVEL…
  - …READ UNCOMMITTED
    allows TA to read dirty data
  - …READ COMMITTED
    forbids dirty reads, but allows TA to issue query several times & get different results (as long as TAs that wrote them have committed)
  - …REPEATABLE READ
    ensures that any tuples will be the same under subsequent reads.
    However a query may turn up new (phantom) tuples
  - …SERIALIZABLE
    default; can be omitted

Effects of New Isolation Levels

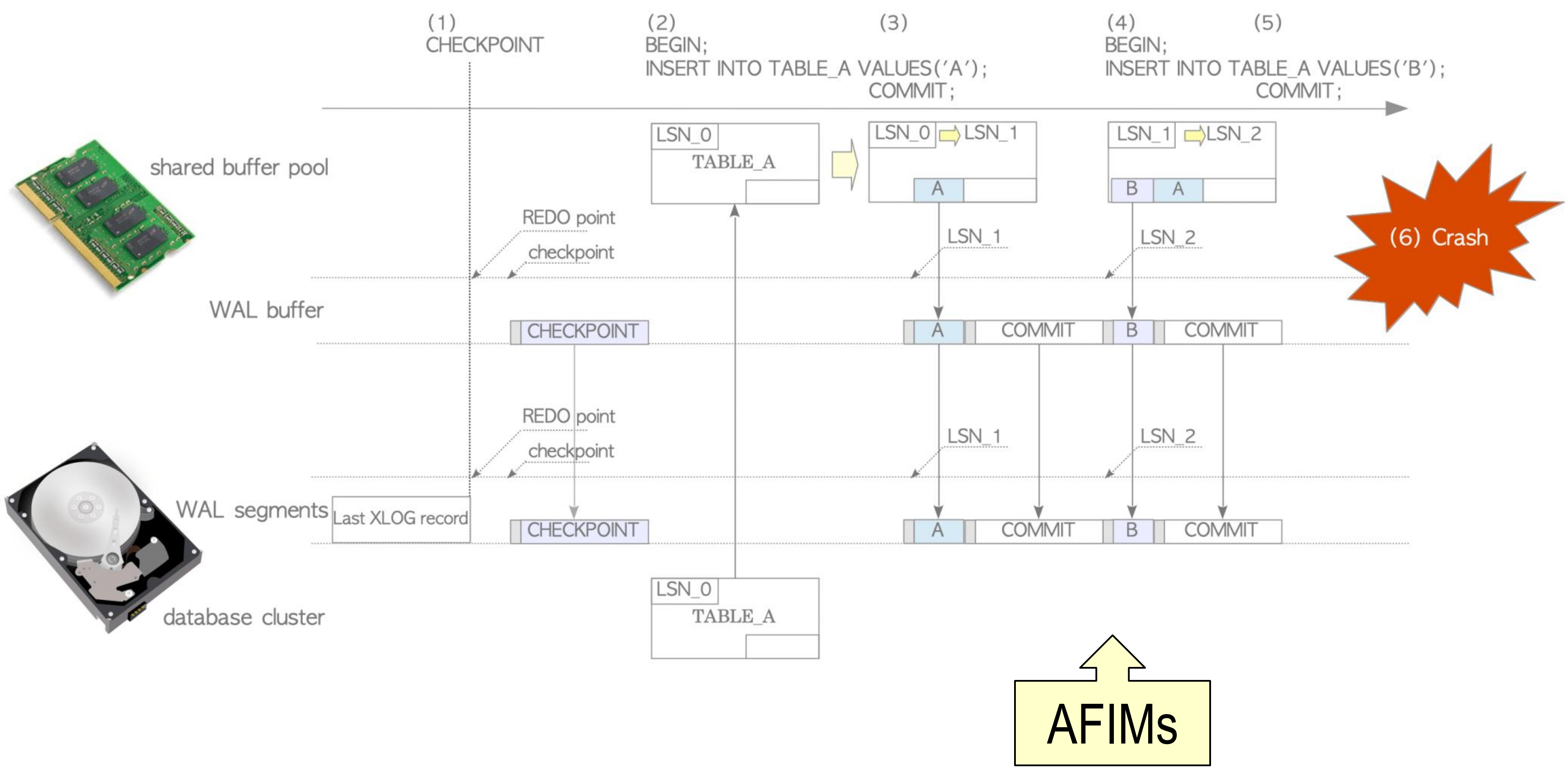
- Consider seat choosing algorithm:
- If run at level READ COMMITTED
  - will not see seats as booked if reserved but not committed (roll back if over-booked)
  - Repeated queries may yield different seats (other TAs booking in parallel)
- If run at REPEATABLE READ
  - any seat found remains available on reload
  - new tuples seen by new queries (e.g. switching to larger plane)



Write-Ahead Logging (WAL)

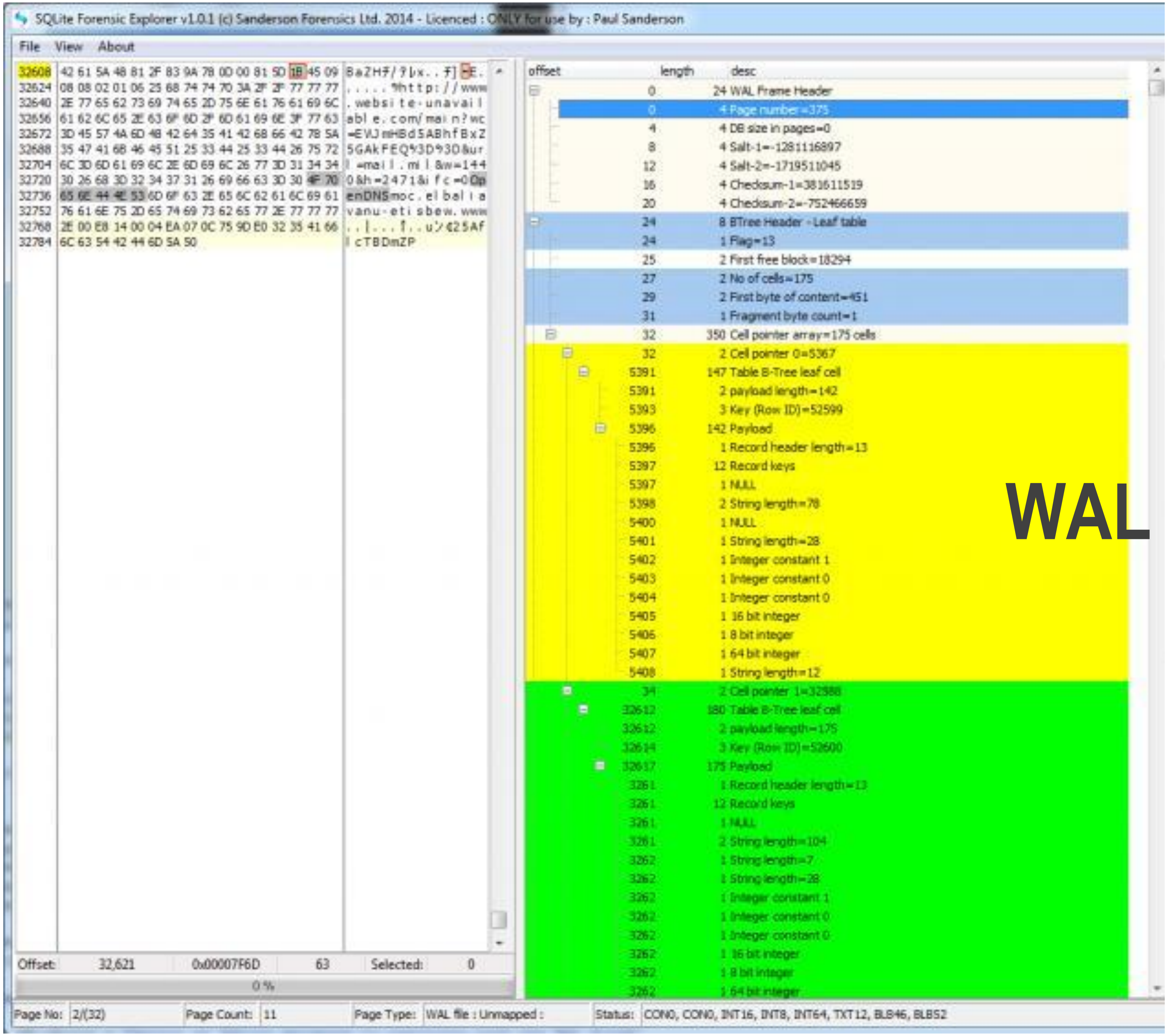
- All change actions recorded in log file(s)
  - Not single tuples, but complete pages affected
  - Before-Image (BFIM) + After-Image (AFIM) allow choice of redo or undo
  - Ti writes an object: TA identifier + BFIM + AFIM
  - Ti commits/aborts: TA identifier + commit/abort indicator
  - Log records chained by TA id → easy to undo specific TA
- Log written before database update = "write ahead"
  - Simply append to log file, so fast
- Log is beating heart of DBMS!
  - Use fast storage
  - often duplexed & archived on stable storage

WAL in Action (PostgreSQL)



[www.interdb.jp]

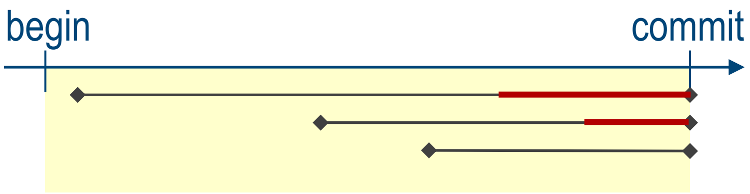
WAL Inspection



[sqliteforensictoolkit.com]

Aborting a Transaction

- If TA Ti is aborted, all its actions have to be undone
  - plus if another Tj reads object last written by Ti, then Tj must be aborted as well!
- Most systems avoid such cascading aborts by releasing TA's locks only at commit time = strict 2PL
  - If Ti writes an object,
    Tj can read this only after Ti commits



- Log serves to find actions to undo when aborting TA

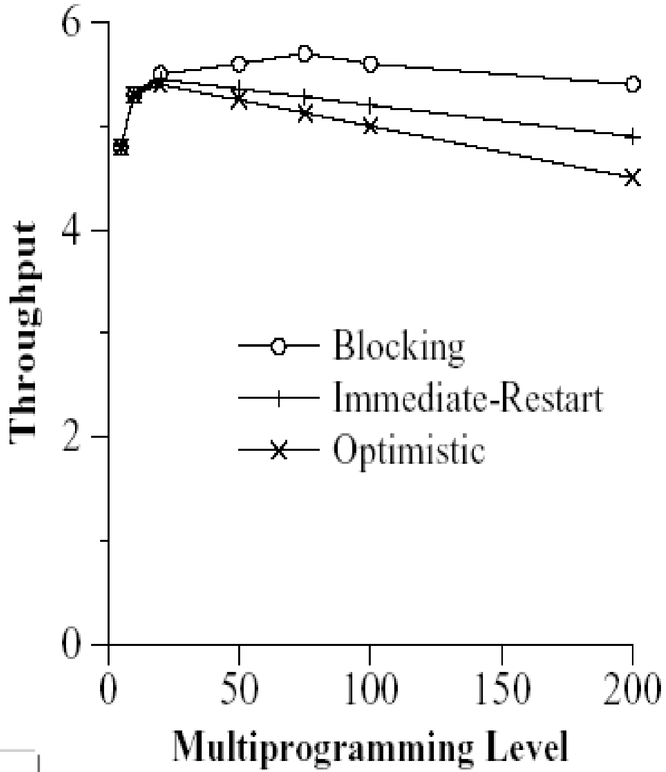
Crash Recovery

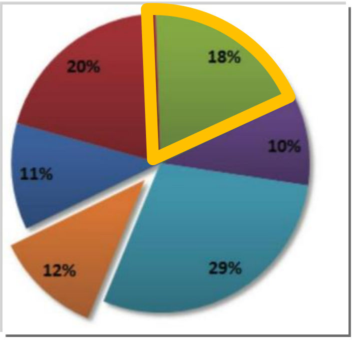
- Log also used to recover from system crashes
  - Abort all TAs active at crash time
  - Re-run changes committed, but not yet permanent at crash time
- Aries recovery algorithm:
  - Analysis: Scan log forward (from most recent checkpoint until crash) to identify
    - *all TAs that were active*
    - *all dirty pages in the buffer pool*
  - Redo: repeat all updates to dirty pages in the buffer pool as needed
    - *to ensure that all logged updates are in fact carried out and written to disk*
  - Undo: nullify writes of all TAs active at crash time working backwards in log
    - *by restoring "before value" of update, which is in log record for update*

Performance Impact

- Lock contention
- Deadlock

- *See NewSQL later!*





Summary

- Concurrency control & recovery: core DBMS functions
  - Safe & reliable data management
  - Concurrency invisible to user
- ACID against update anomalies
- Mechanisms:
  - TA scheduling; Strict 2PL
  - Locks
  - Write-ahead logging (WAL)