



JACOBS
UNIVERSITY

Transaction Management

Ramakrishnan & Gehrke, Chapter 14+

Instructor: Peter Baumann

email: p.baumann@jacobs-university.de

tel: -3178

office: room 88, Research 1

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
⇒ 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
- 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

	S	X
S	+	-
X	-	-

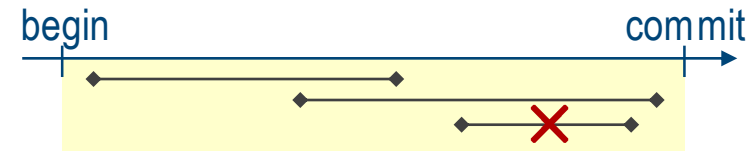
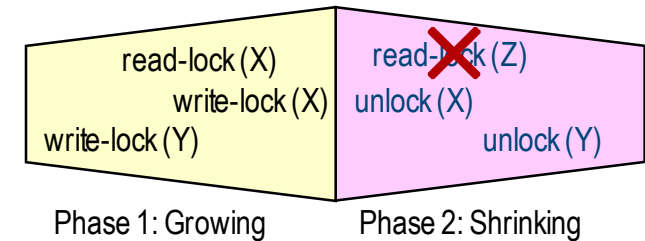
Two-Phase Locking Protocol

■ 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 available seat, **reserve** by setting **occ** to **TRUE**; if there is none, abort*
 - *Ask customer for approval. If so, commit, otherwise release seat by setting **occ** to **FALSE**, goto 1*
 - 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

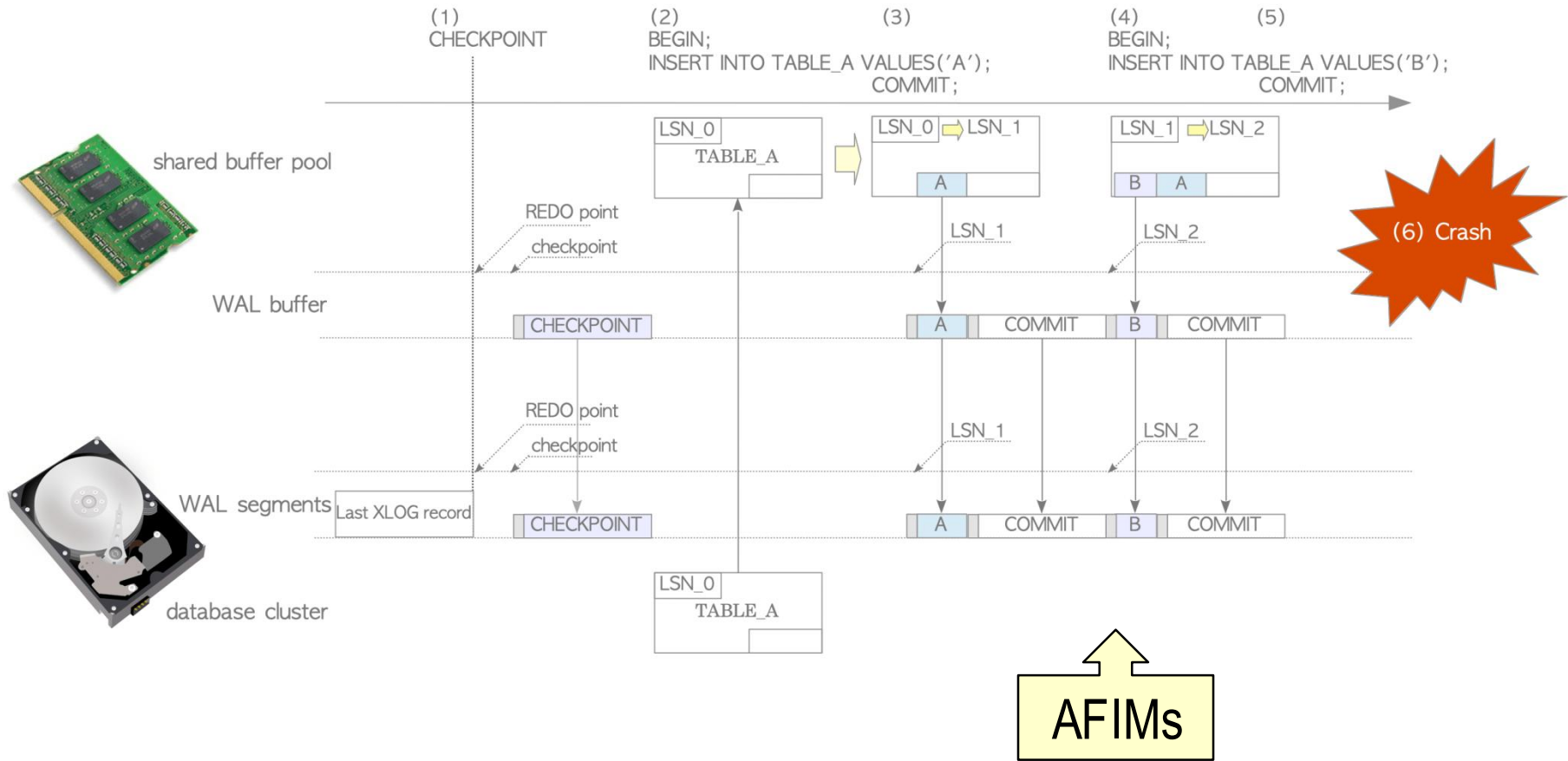
Effects of New Isolation Levels

- Consider seat choosing algorithm:
- If run at level **READ COMMITTED**
 - seat choice function 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 in step 1 will remain available in subsequent queries
 - new tuples entering relation (e.g. switching flight to larger plane) seen by new queries

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: TAidentifier + BFIM + AFIM
 - Ti **commits/aborts**: TAidentifier + commit/abort indicator
 - Log records **chained by TAid** → 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)



File View About

32608 42 61 5A 48 81 2F 83 9A 78 0D 00 81 5D 1B 45 09 8a2H7/9lx..f] E.
32624 08 08 02 01 06 25 68 74 74 70 3A 2F 2F 77 77 77http://www
32640 2E 77 65 62 73 69 74 65 2D 75 6E 61 76 61 69 6C .website-unavail
32656 61 62 6C 65 2E 63 6F 6D 2F 6D 61 69 6E 2F 77 63 abl.e.com/main?wc
32672 3D 45 57 4A 6D 48 42 64 35 41 42 68 66 42 78 5A =EVJnHsdSABhfBx2
32688 35 47 41 68 46 45 51 25 33 44 25 33 44 26 75 72 5GakFEQ93D9308ur
32704 6C 3D 6D 61 69 6C 2E 6D 69 6C 26 77 3D 31 34 34 l=mail.mil&n=144
32720 30 26 68 3D 32 34 37 31 26 69 66 63 3D 30 4F 70 08h=2471&f c=0Op
32736 65 6E 44 4E 53 6D 6F 63 2E 65 6C 62 61 6C 69 61 endDNSmoc.elbalia
32752 76 61 6E 75 2D 65 74 69 73 62 65 77 2E 77 77 77 vanu-etisbew.www
32768 2E 00 E8 14 00 04 EA 07 0C 75 9D E0 32 35 41 66 ...l...f...u/425Af
32784 6C 63 54 42 44 6D 5A 50 cTBDmZP

offset	length	desc
0	24	WAL Frame Header
0	4	Page number=375
4	4	DB size in pages=0
8	4	Salt-1=-128116897
12	4	Salt-2=-171951045
16	4	Checksum-1=381611519
20	4	Checksum-2=752466659
24	8	BTree Header - Leaf table
24	1	Flag=13
25	2	First free block=18294
27	2	No of cells=175
29	2	First byte of content=451
31	1	Fragment byte count=1
32	350	Cell pointer array=175 cells
32	2	Cell pointer 0=5367
5391	147	Table B-Tree leaf cell
5391	2	payload length=142
5393	3	Key (Row ID)=52599
5396	142	Payload
5396	1	Record header length=13
5397	12	Record keys
5397	1	NULL
5398	2	String length=78
5400	1	NULL
5401	1	String length=28
5402	1	Integer constant 1
5403	1	Integer constant 0
5404	1	Integer constant 0
5405	1	16 bit integer
5406	1	8 bit integer
5407	1	64 bit integer
5408	1	String length=12
34	2	Cell pointer 1=32888
32612	180	Table B-Tree leaf cell
32612	2	payload length=175
32614	3	Key (Row ID)=52600
32617	175	Payload
3261	1	Record header length=13
3261	12	Record keys
3261	1	NULL
3261	2	String length=104
3262	1	String length=7
3262	1	String length=28
3262	1	Integer constant 1
3262	1	Integer constant 0
3262	1	Integer constant 0
3262	1	16 bit integer
3262	1	8 bit integer
3262	1	64 bit integer

Offset: 32,621 0x00007F6D 63 Selected: 0

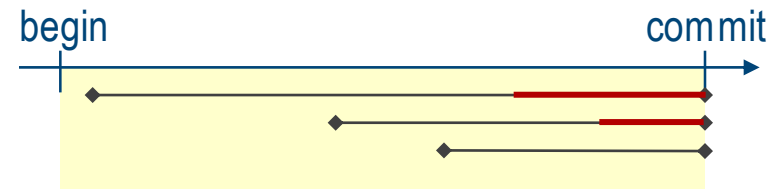
0 %

Page No: 2/(32) Page Count: 11 Page Type: WAL file : Unmapped : Status: CONQ, CONQ, INT16, INT8, INT64, TXT12, BLS46, BLS52

WAL Inspection

Aborting a Transaction

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

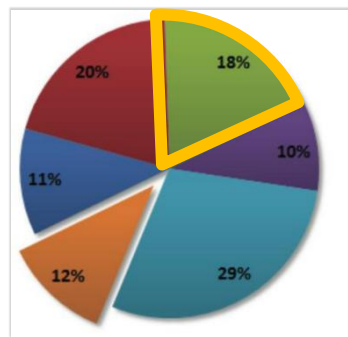
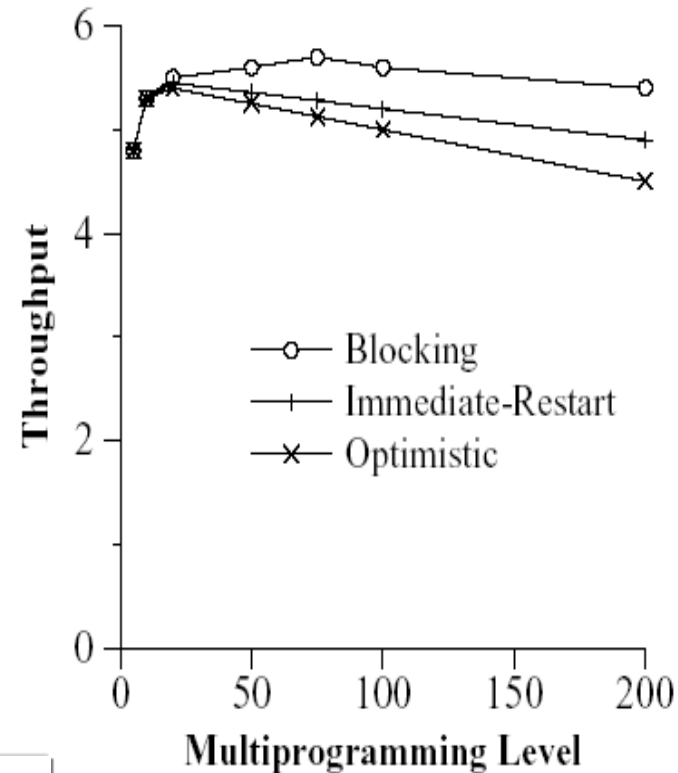


Crash Recovery

- 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)