Ch 16. Transaction Management: Overview

Sang-Won Lee

http://icc.skku.ac.kr/~swlee

SKKU VLDB Lab.

(http://vldb.skku.ac.kr/)

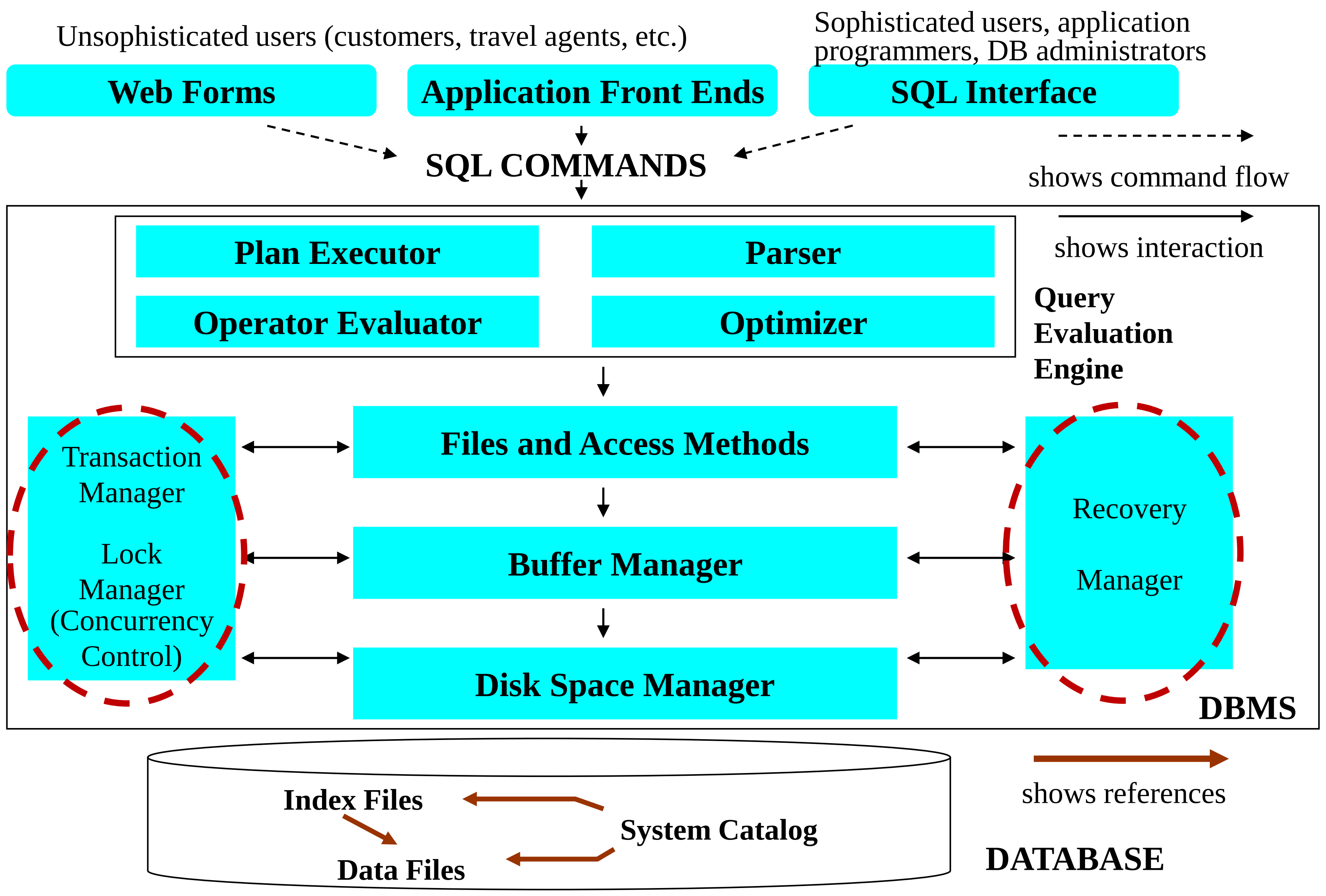


Figure 1.3 Anatomy of an RDBMS

OLTP Through the Looking Glass [sigmod 08]

New Order Transaction

1. Select(whouse-id) from Warehouse
2. Select(dist-id, whouse-id) from District
3. Update(dist-id, whouse-id) in District
4. Select(customer-id, dist-id, whouse-id) f
5. Insert into Order
6. Insert into New-Order
7. For each item (10 items):
   (a) Select(item-id) from Item
   (b) Select(item-id,whouse-id) from Stock
   (c) Update(item-id,whouse-id) in Stock
   (d) Insert into Order-Line
8. Commit

New Order

```
begin
for loop(10)
.....Btree lookup(I), pin
Btree lookup(D), pin
Btree lookup (W), pin
Btree lookup (C), pin
update rec (D)
for loop (10)
.....Btree lookup(S), pin
.....update rec (S)
.....create rec (O-L)
.....insert Btree (O-L)
create rec (O)
insert Btree (O)
create rec (N-O)
insert Btree (N-O)
insert Btree 2ndary(N-O)
commit
```

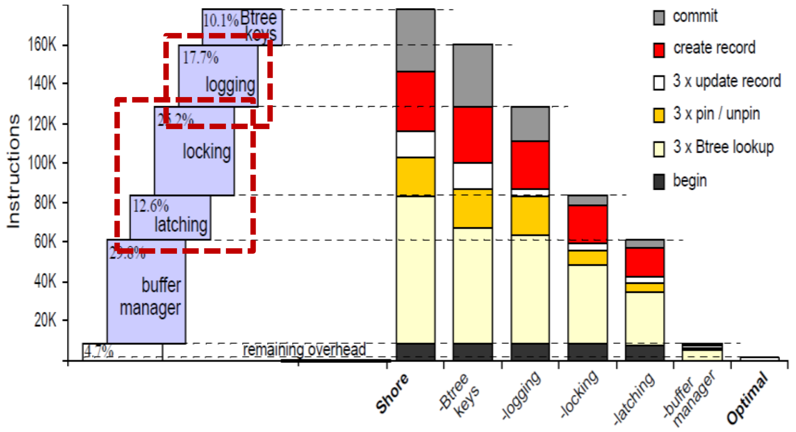


Figure 5. Detailed instruction count breakdown for Payment transaction.

Relation	New Order	Payment	Order Status	Delivery	Stock Level	Average
warehouse	U(1)	U(1)				0.87
district	U(1)	U(1)			P(1)	0.93
customer	NU(1)	NU(2.2)	NU(2.2)	P(10)		1.524
stock	NU(10)				P(200)	12.4
item	NU(10)					4.4
order	A(1)		P(1)	P(10)		0.53
new-order	A(1)			P(10)		0.49
order-line	A(10)		P(10)	P(100)	P(200)	13.3
history		A(1)				0.43

A Sample Transaction Program

```
ACCOUNT(ACCOUNT_NUMBER, CUSTOMER_NUMBER, ACCOUNT_BALANCE, HISTORY)
CUSTOMER(CUSTOMER_NUMBER, CUSTOMER_NAME, ADDRESS,.....)
HISTORY(TIME, TELLER, CODE, ACCOUNT_NUMBER, CHANGE, PREV_HISTORY)
CASH_DRAWER(TELLER_NUMBER, BALANCE)
BRANCH_BALANCE(BRANCH, BALANCE)
TELLER(TELLER_NUMBER, TELLER_NAME,......)
```

From Gray's Presentation

```
exec sql begin declare section;
long Aid, Bid, Tid, delta, Abalance;
exec sql end declare section;
DCApplication()
{    read input msg;
     exec sql begin transaction;
     Abalance = DoDebitCredit(Bid, Tid, Aid, delta);
     send output msg;
     exec sql commit transaction; }
```

A transaction
= A sequence of SQL DML statements
= A sequence of Reads and Writes
(from/to records in pages)

```
Long DoDebitCredit(long Bid, long Tid, long Aid, long delta)
{    exec sql update accounts set Abalance = Abalance + :delta where Aid = :Aid;
     exec sql select Abalance into :Abalance from accounts where Aid = :Aid;
     exec sql update tellers  set Tbalance = Tbalance + :delta where Tid = :Tid;
     exec sql update branches set Bbalance = Bbalance + :delta  where Bid = :Bid;
     exec sql insert into history(Tid, Bid, Aid, delta, time)  values (:Tid, :Bid, :Aid, :delta, CURRENT);
return(Abalance); }
```

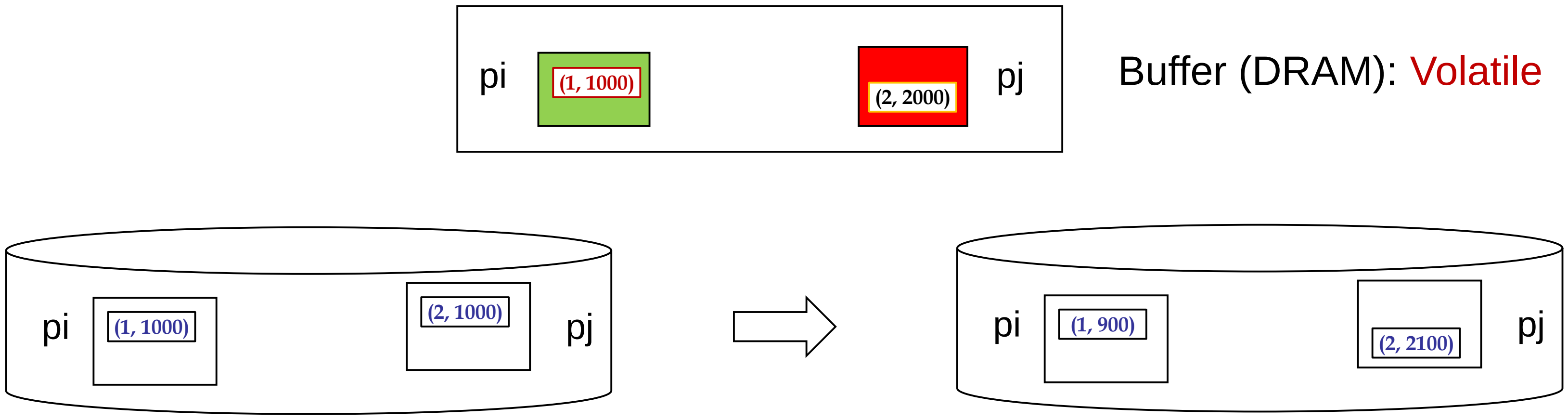
Sample Transaction and DB Architectural Interpretation

```
create table account (id, balance);
insert into account values (1, 1000);
insert into account values (2, 2000);
commit;
```

```
-- TX1: transfer 100$ from Acct# 1 to Acct# 2:
BeginTX;
update account set balance = balance - 100 where id = 1;
update account set balance = balance + 100 where id = 2;
Commit;  /* Or, rollback or killed */
```

TX1: A=A-100 B=B+100

TX1: BeginTX Read Pi, Write Pi, Read Pj, Write Pj, Commit

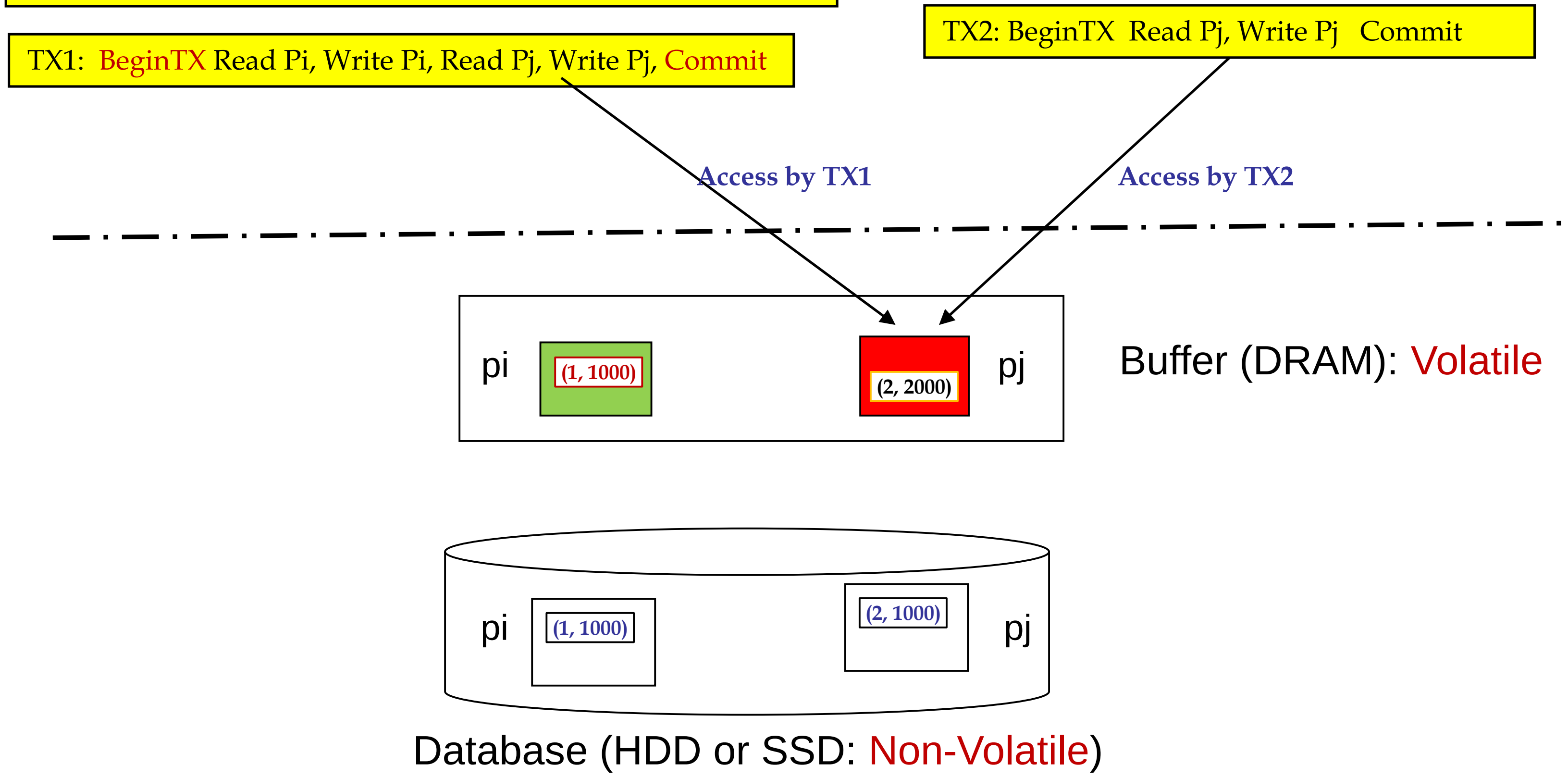


Database (HDD or SSD: Non-Volatile)

Sample Transaction and DB Architectural Interpretation

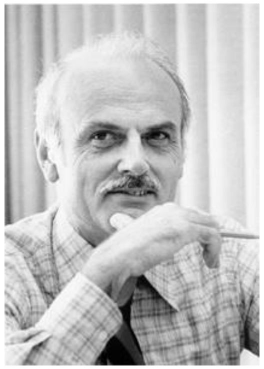
```
-- TX1: transfer 100$ from Acct# 1 to Acct# 2:
Begin_TX;
update account set balance = balance - 100 where id = 1;
update account set balance = balance + 100 where id = 2;
Commit;
```

```
-- TX2: increase account 2's balance by 200;
Begin_TX;
update account set balance = balance + 200 where id = 2;
commit;
```


TX1: BeginTX Read Pi, Write Pi, Read Pj, Write Pj, Commit
TX2: BeginTX Read Pj, Write Pj Commit
Access by TX1
Access by TX2
pi
(1, 1000)
(2, 2000)
pj
Buffer (DRAM): Volatile
pi
(1, 1000)
(2, 1000)
pj
Database (HDD or SSD: Non-Volatile)

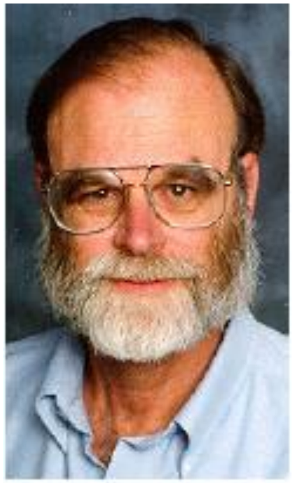
Core Technology of RDBMS

- Relational data model
  - E. F. Codd: 1980 winner of Turing Award



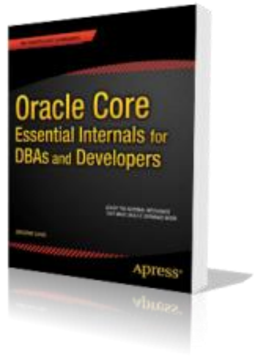
- Query optimizer

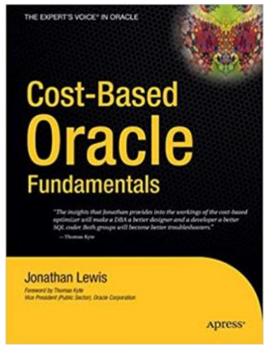
- Transaction management
  - Jim Gray: 1998 winner of Turing Award



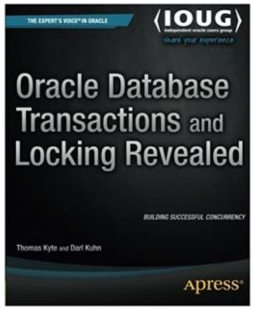
On Oracle Transaction Mechanism: Recommended Books

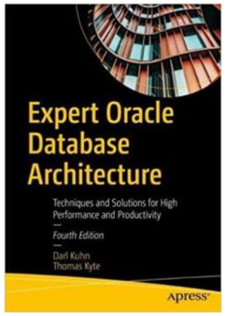
- Jonathan Lewis:
  https://jonathanlewis.wordpress.com/





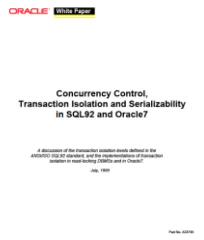
- Tom Kyte:
  https://asktom.oracle.com/


THE EXPERT'S VOICE® IN ORACLE
IOUG
Oracle Database Transactions and Locking Revealed
BUILDING SUCCESSFUL CONCURRENCY
Thomas Kyte and Darl Kuhn
Apress


Expert Oracle Database Architecture
Techniques and Solutions for High Performance and Productivity
Fourth Edition
Darl Kuhn
Thomas Kyte
Apress

- Ken Jacobs

"Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7" Oracle White Paper



ORACLE White Paper

Concurrency Control,
Transaction Isolation and Serializability
in SQL92 and Oracle7

Transaction

- Our world: 1) **unreliable** computers and 2) **concurrent** executions
  - Can cause numerous problems

- The notion of transaction = one of the 'big ideas' in CS
  - **What if** programmer is responsible for concurrency control (CC) and recovery? e.g. file system
  - Awesomely powerful **abstraction** to cope with those problems
  - If DBMS can handle **CC and recovery** **transparently** while DB appl. programmer focuses on the **logic of each transaction** without explicit concern for CC and failure, **programmer productivity** increases tremendously.
- Transaction
  - Unit of work ← Developer perspective
  - **Unit of CC and Recovery** ← DBMS engine perspective

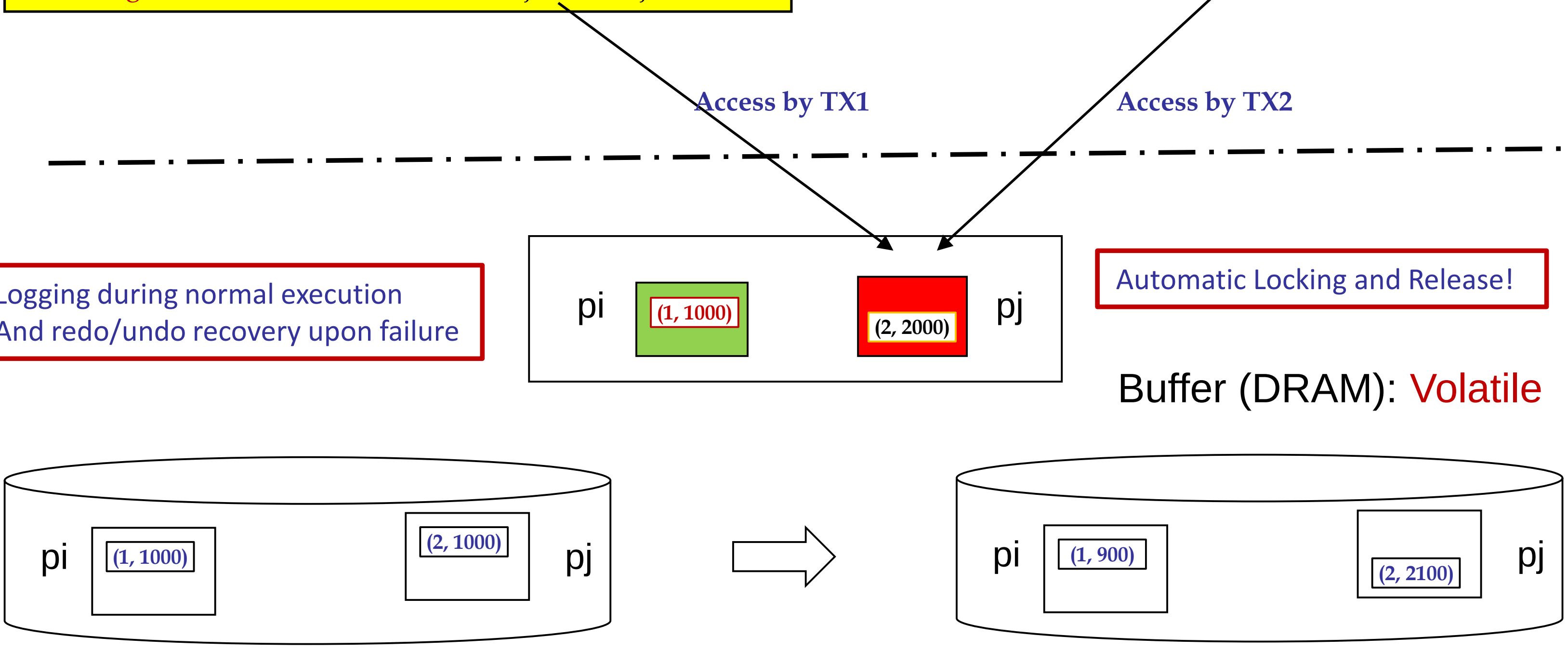
Transaction: Automatic Recovery and Concurrency

-- TX1: transfer 100$ from Acct# 1 to Acct# 2:
Begin_TX;
update account set balance = balance - 100 where id = 1;
update account set balance = balance + 100 where id = 2;
Commit;

-- TX2: increase account 2's balance by 200;
Begin_TX;
update account set balance = balance + 200 where id = 2;
commit;

TX1: BeginTX Read Pi, Write Pi, Read Pj, Write Pj, Commit

TX2: BeginTX Read Pj, Write Pj Commit



Logging during normal execution
And redo/undo recovery upon failure

Automatic Locking and Release!

Buffer (DRAM): Volatile

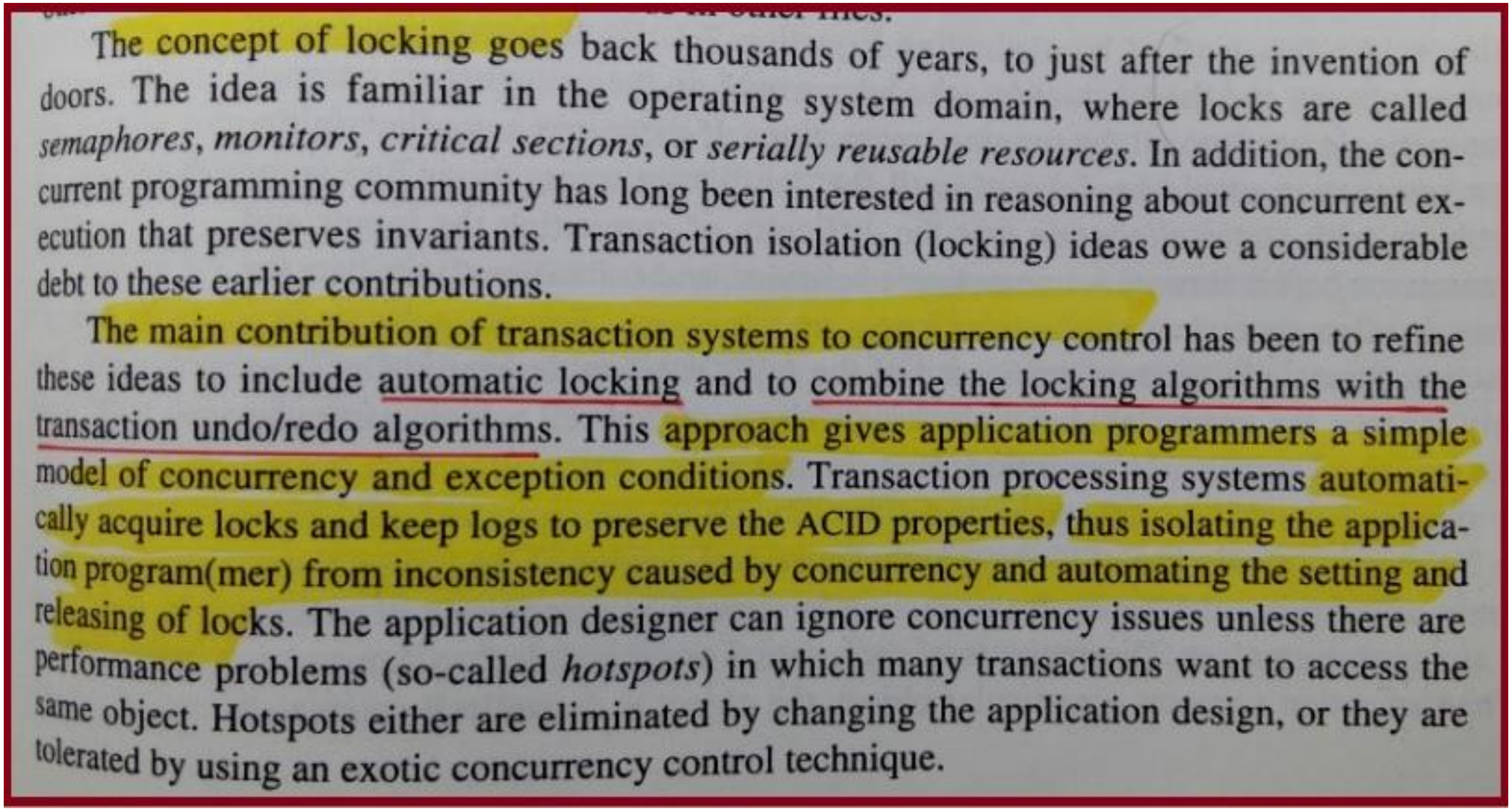
Database (HDD or SSD: Non-Volatile)

Terminology

- Isolation = generic
- Concurrency control = problem
- Serializability = theory
- Locking = an implementation technique

(source: Jim Gray’s TP book)

Contribution of Automatic Locking (Gray's TP Book)



The concept of locking goes back thousands of years, to just after the invention of doors. The idea is familiar in the operating system domain, where locks are called *semaphores*, *monitors*, *critical sections*, or *serially reusable resources*. In addition, the concurrent programming community has long been interested in reasoning about concurrent execution that preserves invariants. Transaction isolation (locking) ideas owe a considerable debt to these earlier contributions.

The main contribution of transaction systems to concurrency control has been to refine these ideas to include automatic locking and to combine the locking algorithms with the transaction undo/redo algorithms. This approach gives application programmers a simple model of concurrency and exception conditions. Transaction processing systems automatically acquire locks and keep logs to preserve the ACID properties, thus isolating the application program(mer) from inconsistency caused by concurrency and automating the setting and releasing of locks. The application designer can ignore concurrency issues unless there are performance problems (so-called *hotspots*) in which many transactions want to access the same object. Hotspots either are eliminated by changing the application design, or they are tolerated by using an exotic concurrency control technique.

Big Picture: CC & R

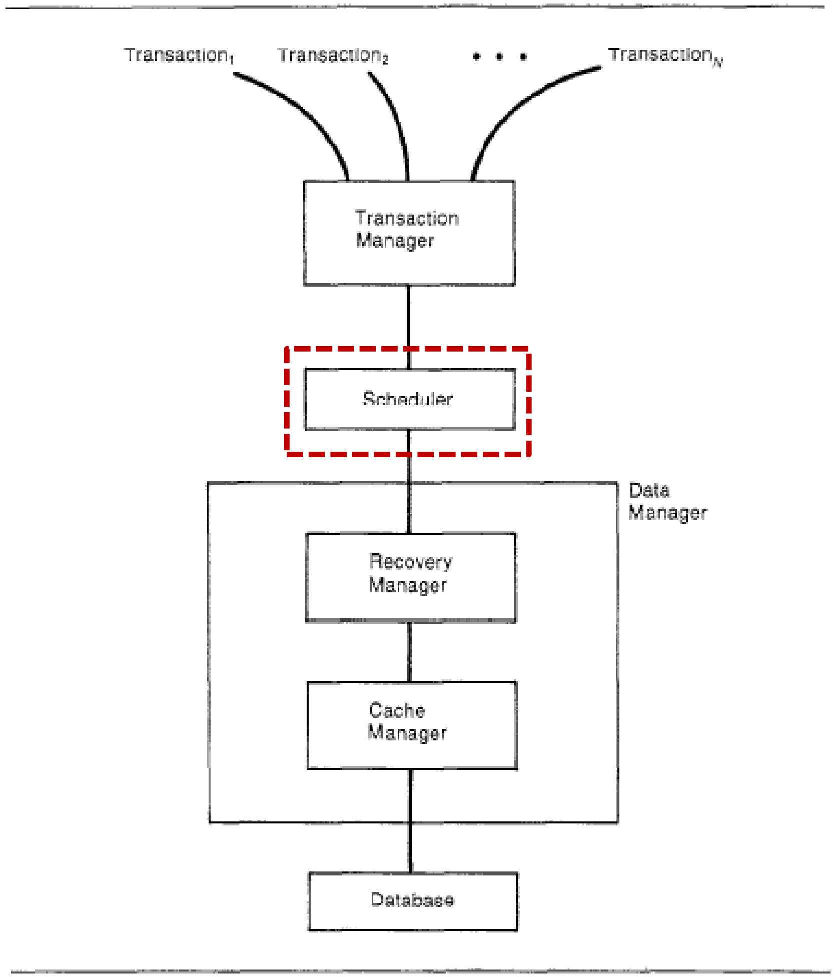


FIGURE 1–1
Centralized Database System

Source: Bernstein et. al.

What you should know from this chapter

- What is transaction?
  - 4 properties of transactions
- Why TRs should execute interleavingly?
- Criteria for correct interleaved execution: serializable schedule
- Anomalies from concurrent executions?
- How a DBMS use locks for correctness?
  - Impact of locking on performance
- CC in SQL

16.1 Transaction

- Definition (business-side): **A unit of works**
  - Our daily life is full of transactions: e.g., class registration, money transfer, ticketing/booking, online/offline shopping, etc.
  - e.g. Money transfer: Account A → Account B: 100$
    - ✓ Begin_TX; A = A – 100$; B = B + 100$; Commit or Rollback;
    - ✓ The fundamental property: "All-or-nothing"
    - ✓ Transition from a normal state to another normal state

Transaction(2)

- Definition (technical-side): the foundation of concurrent execution and recovery from unreliable HW/SW failures

Business-side Properties	Technical Properties of Transaction	Technical Threats	Responsible Subject
All-or-nothing	**A**tomicity	System Failures	Recovery mgr.
Consistency	**C**onsistency	Errors in Transaction Logics	Appl. Programmers
	Isolation	Concurrent Exec.	CC mgr.
	Durability	System Failures	Recovery mgr.

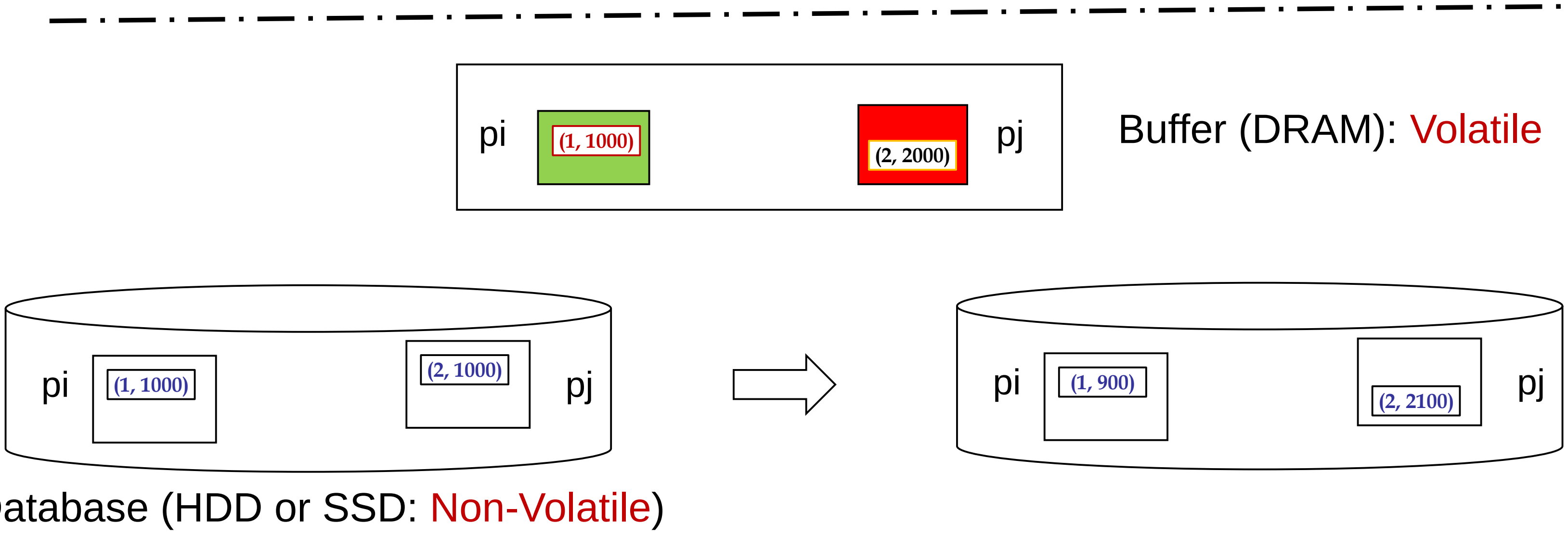
4 Properties of transactions and Responsible subject to guarantees them?

ACID Properties

- Atomicity
  - Users should be able to regard the execution of each transaction as atomic: ALL-OR-NOTHING
- Consistency
  - Each transaction, run by itself, must preserve the consistency of the database
- Isolation
  - Each transaction is isolated or protected from the effects of concurrent other transactions
- Durability
  - Once completed, a transaction’s effect should persist in spite of system crashes

ACID Properties (2): Atomicity, Consistency, and Durability

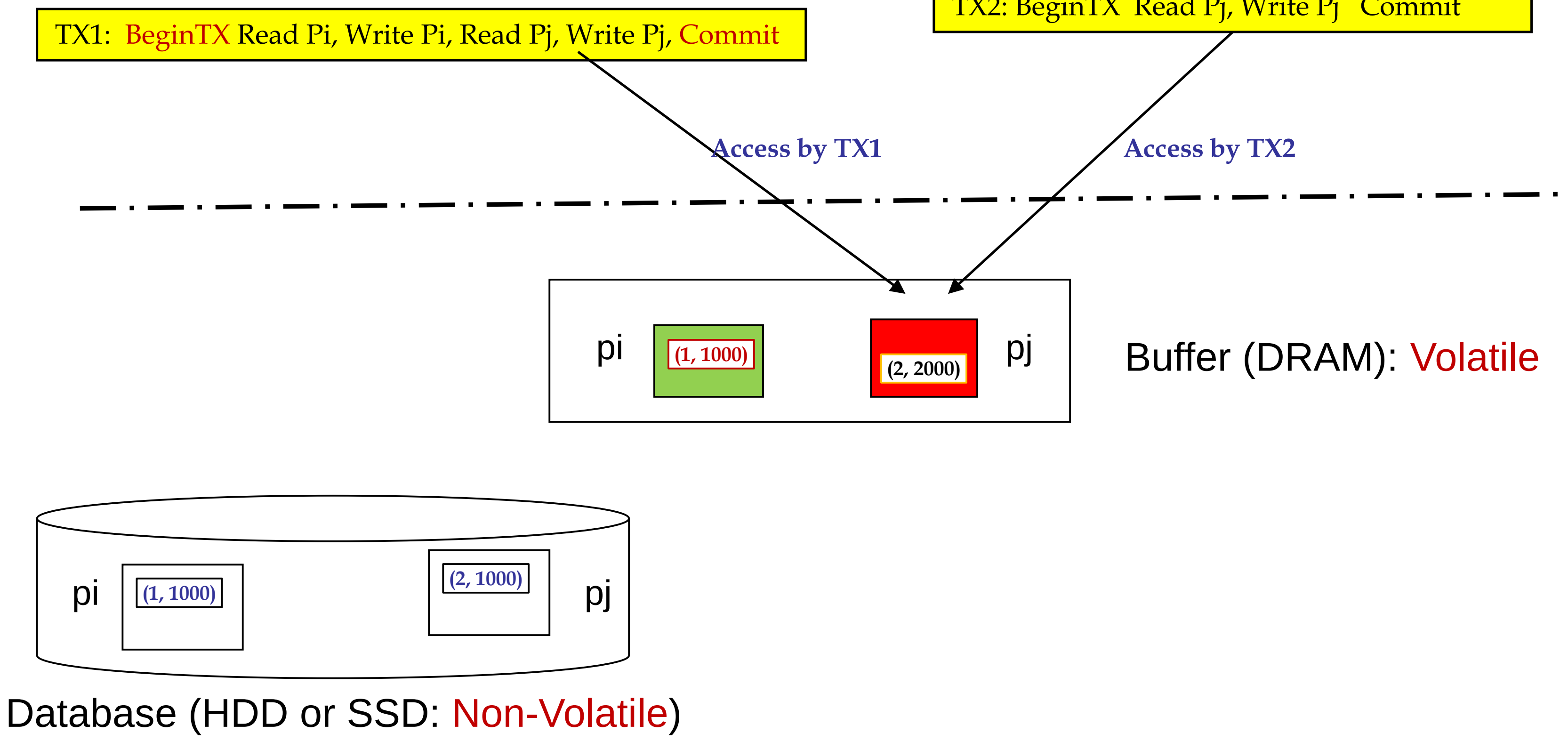
```
-- TX1: transfer 100$ from Acct# 1 to Acct# 2:
BeginTX;
update account set balance = balance - 100 where id = 1;
update account set balance = balance + 100 where id = 2;
Commit;        /* Or, rollback or killed */
```



ACID Properties (3): Isolation

```
-- TX1: transfer 100$ from Acct# 1 to Acct# 2:
Begin_TX;
update account set balance = balance - 100 where id = 1;
update account set balance = balance + 100 where id = 2;
Commit;
```

```
-- TX2: increase account 2's balance by 200;
Begin_TX;
update account set balance = balance + 200 where id = 2;
commit;
```


TX1: BeginTX Read Pi, Write Pi, Read Pj, Write Pj, Commit
TX2: BeginTX Read Pj, Write Pj Commit
Access by TX1
Access by TX2
pi
(1, 1000)
(2, 2000)
pj
Buffer (DRAM): Volatile
pi
(1, 1000)
(2, 1000)
pj
Database (HDD or SSD: Non-Volatile)

ACID Properties (4)

- Why incomplete transaction?
  1. explicit abort by user
  2. system crashes
  3. aborted by DBMS engine, e.g. due to dead lock; in this case, automatic restart

- To ensure "all-or-nothing" for incomplete TRs,
  - DBMS should be able to "UNDO" the actions by the TRs
  - for this, DBMS maintains "LOG" of all writes to database

- The "LOG" is also used to ensure durability
  - if the system crashes before the changes made by a complete transaction are written to disk, the log is used to remember and restore these changes when the system restarts

Three Types of Transaction Life

BEGIN WORK
Operation 1
Operation 2
…
Operation k
COMMIT WORK

Successful completion
< 96% of all transactions

BEGIN WORK
Operation 1
Operation 2
…
(I got lost !)
ROLLBACK WORK

Application requests
termination (Suicide)
> 3% of all transactions

BEGIN WORK
Operation 1
Operation 2
…
Rollback due to external
cause like timeout etc.

Forced termination
(Murder)
> 1% of all transactions

From Gray's Presentation

16.2 Transactions and Schedules

- A *transaction* is the DBMS's abstract view of a user program: a sequence of actions – begin, reads, writes, commit, and abort

- A *schedule* is a list of actions from a set of transactions
  - Actions from different/concurrent transactions **may interleave**
  - BUT! actions from a transaction **should preserve** their order in the transaction

```
Long DoDebitCredit(long Bid, long Tid, long Aid, long delta)
{    exec sql update accounts set Abalance = Abalance + :delta where Aid = :Aid;
     exec sql select Abalance into :Abalance from accounts where Aid = :Aid;
     exec sql update tellers  set Tbalance = Tbalance + :delta where Tid = :Tid;
     exec sql update branches set Bbalance = Bbalance + :delta  where Bid = :Bid;
     exec sql insert into history(Tid, Bid, Aid, delta, time)  values (:Tid, :Bid, :Aid, :delta, CURRENT);
return(Abalance); }
```

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(B)$
	$W(B)$
$R(C)$	
$W(C)$	

Figure 16.1

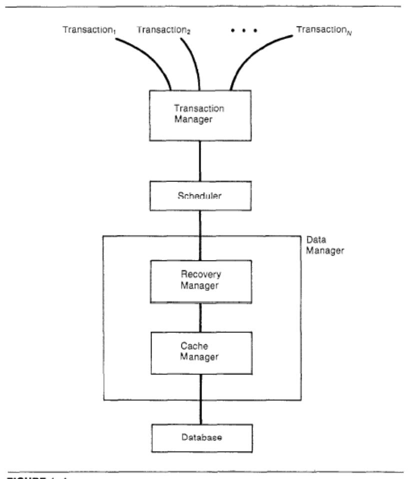


FIGURE 1–1
Centralized Database System

16.3 Concurrent Execution of Transactions

- Concurrent exec. of user programs is essential for performance!
  - Throughput: because disk accesses are frequent, and relatively slow, it is important to keep the CPU and I/O overlapping
  - Response time: when a short transaction execute with a long transaction, interleaved execution allows better response time for the short one

Concurrent Schedule: Example

- Consider two transactions:

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

  - T1: transferring $100 from B's account to A's account
  - T2: crediting both accounts with a 6% interest payment.

- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.
  - T1 --> T2; or T2 --> T1;

Example (Contd.)

- Consider a possible interleaving (*schedule*): A, B are records.

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)	

16.3.2 Serializability:
Criteria for Correct Concurrent Transactions

- Serial schedule: schedule that does not interleave the actions of different transactions.

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

- 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.

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

- Serializable schedule: A schedule that is equivalent to some serial execution of the transactions.
  - Note: if each transaction preserves consistency, every serializable schedule preserves consistency.

16.3.3 Anomalies with Interleaved Execution

- Reading Uncommitted Data (Dirty Read, W-R Conflicts)

T1:	R(A), W(A),		R(B), W(B), **Abort**
T2:		R(A), W(A), C	

- Unrepeatable Reads (R-W Conflicts):

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

- Overwriting Uncommitted Data (Dirty Write, W-W Conflicts)

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

16.4 Lock-Based Concurrency Control

- We use "locks" to control access to items.
  - Shared (S) locks – multiple TXs can hold these on a particular item concurrently
  - Exclusive (X) locks – only one of these and no other locks, can be held on a particular item at a time.

	R(S)	W(X)
R(S)	C	X
W(X)	X	X

Lock Compatibility Table
- S: shared lock
- X: exclusive lock
- C: compatible
- X: non-compatible

- Lock manager: 1) (un)locking requests, 2) (un)blocking TXs

```
T1:        R(A), W(A),                        R(B), W(B), Abort
T2:                          R(A), W(A), C
```

- Lock-based CC: pessimistic
  - cf. Optimistic concurrency control. (OCC)

From M. Franklin's Presentation

Very Large Data Bases

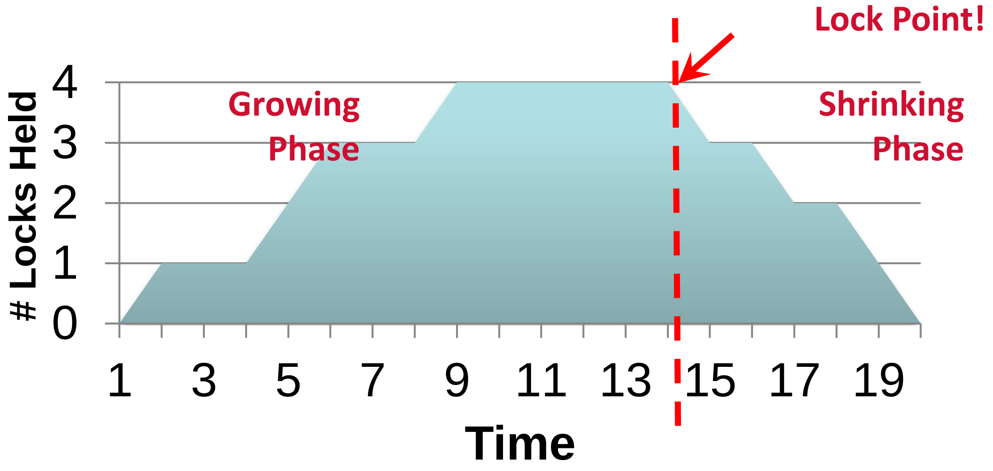
16.4 Lock-Based Concurrency Control

(Assumption: **"single version"** concurrency control)

- Two-phase locking (2PL) protocol

  1. Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  2. A TX can not request additional locks once it releases any locks.

→ Each TX has a "growing phase" followed by a "shrinking phase".



From M. Franklin's Presentation

16.4 Lock-Based Concurrency Control

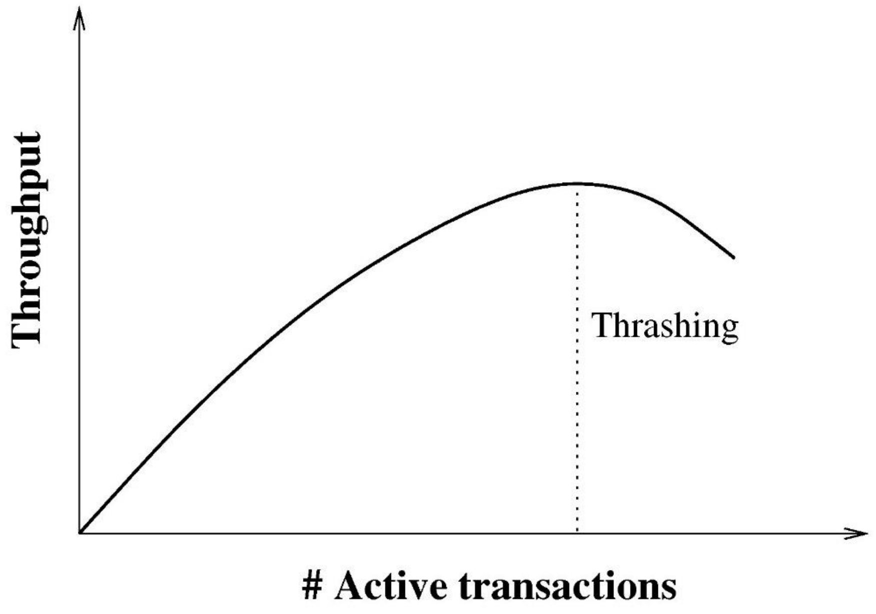
```
T1:  R(A), W(A),   ...................... Abort
T2:                       R(A), W(A) C
```

- 16.3.4: Abort and unrecoverable schedule
  - See Figure 16.5.
    - ✓ If T1 abort, T2 should be also aborted: cascaded rollback
    - ✓ And, unrecoverable!
  - Thus, holding **all w-locks** until EOT: Strict 2PL

- 16.4.1: ***Strong Strict*** two-phase locking:
  - **All read/write locks** by TXi are held until EOT of TXi (commit/abort):
    - ✓ In practice, we don't know how TX is going to execute when it will end
    - ✓ long vs. short duration lock
  - Strong Strict 2PL allows only serializable schedules.

- 16.4.2: Deadlock: e.g. timeout and victim
  - Oracle: https://docs.oracle.com/cd/B28359_01/server.111/b28318/consist.htm#CNCPT1336

16.5 Lock Thrashing



- See sec 20.10 on how to improve concurrency

16.6 Transaction Support in SQL

- DDL & DML
- Commit/Rollback
- Savepoint: Nested Transaction
  - SAVEPOINT <name>
  - ROLLBACK TO SAVEPOINT <name>

Very Large Data Bases

16.6.2 Transaction Support in SQL

```
SELECT  S.rating, MIN(S.age)
FROM    Sailors S
WHERE  S.rating = 8
```

- Lock granularity
  - What should we lock?
    - ✓ e.g. database, file, subfiles, table, page, row, field
    - ✓ e.g. tuple level: lock bit in each record in Oracle (refer to Ch. 9)
  - Trade-off between concurrency and overhead
    - ✓ Tuple-level locking: lock table, 100 instructions per a lock
    - ✓ Multi-granularity locking, lock escalation [Gray 75]
  - File system vs. DBMS; lock granularity vs. IO unit

- "Phantom problem"
  - What if a new record with "rating = 8" is inserted?
  - Item lock is not enough. Thus, predicate lock need to be introduced
  - Predicate lock has run-time overhead; key-range lock, multi-granule lock;
  - cf. Oracle uses commitSCN of each record to filter out "phantom tuple" which is inserted or updated after a transaction had started

16.6.3 Transaction Support in SQL

- Access mode: READ ONLY; READ WRITE
- Isolation level: **WHY several levels??? Good for lock-based CC implementation**

Level	Dirty Write	Dirty Read	Non-Repeatable Read	Phantom Read
Read Uncommitted (W blocks W, but W does not block R)	N	Y	Y	Y
Read Committed (W blocks R)	N	N	Y	Y
Repeatable Read (R blocks W)	N	N	N	Y
Serializable	N	N	N	N

Figure 16.10 TR Isolation Level in SQL-99

	R(S)	W(X)
R(S)	C	X
W(X)	X	X

Lock Compatibility Table

- S: shared lock
- X: exclusive lock
- C: compatible
- X: non-compatible

- NOTE: **Lock duration** may depend on isolation level
  - In all isolation levels, all write locks are assumed to be long until EOT
  - E.g. Read locks are short duration in RC, while long duration in RR

CC Mechanism in Oracle

- A variant of time-stamp based multi-version conc. control (MVCC)
  - See Ch 17.6.3 for details;
  - See also Oracle data concurrency and consistency

- Key concepts: 1) lock, 2) multi-version, 3) time-stamp based read consistency
  - Only write lock; each object has its write lock
  - Object can have multiple versions with its commitSCN (or clones in buffer)
  - Each SQL statement or each transaction has its timestamp for read consistency

CC Mechanism in Oracle (2)

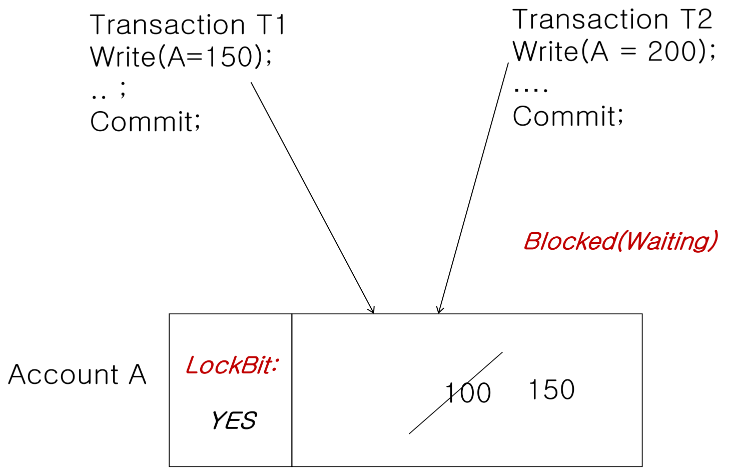
- Transaction commands in Oracle
  - Tx: Begin (can start implicitly!), DMLs , Commit or Abort
  - Two isolation levels: “read committed” and “serializable”
    - ✓ Why not four? Revisit later

```
SQL> set transaction isolation level serializable;        /* by default, read committed */
SQL> alter session set isolation_level = serializable;   /* by default, read committed */
```

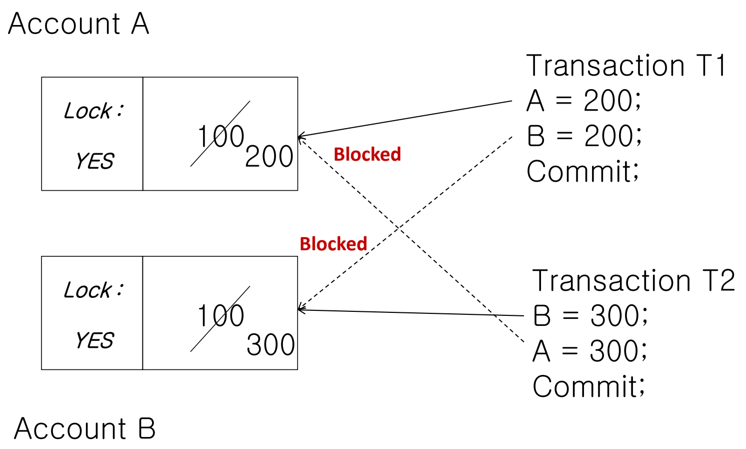
  - A DDL command is a transaction by itself.

CC Mechanism in Oracle (3)

- Write lock
  - Writer still blocks writer
  - Deadlock can happen



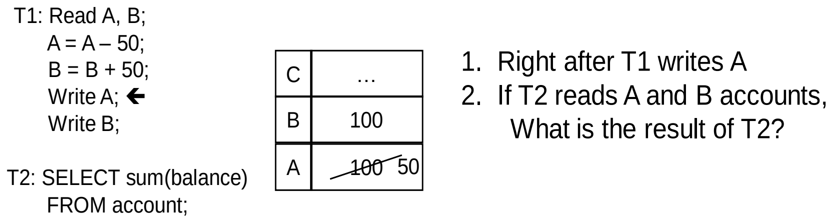
Lock Bit



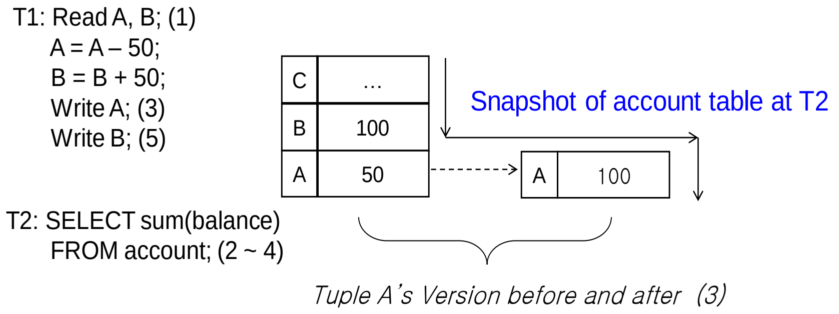
Deadlock

CC Mechanism in Oracle (4)

- BUT!, reader and writer does not block each other.
  - Oracle does not set read lock
  - Multiple versions of the same page can be cloned in buffer pool
  - Thus, no dirty read without "writer-blocks-reader"
  - Also, without "read-blocks-write", repeatable read is possible.
- How to avoid phantom?
  - Each TX has its starting commitSCN timestamp
  - Oracle seems to use commitSCN of each record to filter out "phantom tuples" which are inserted or updated after a transaction had started



A) Dirty Reads

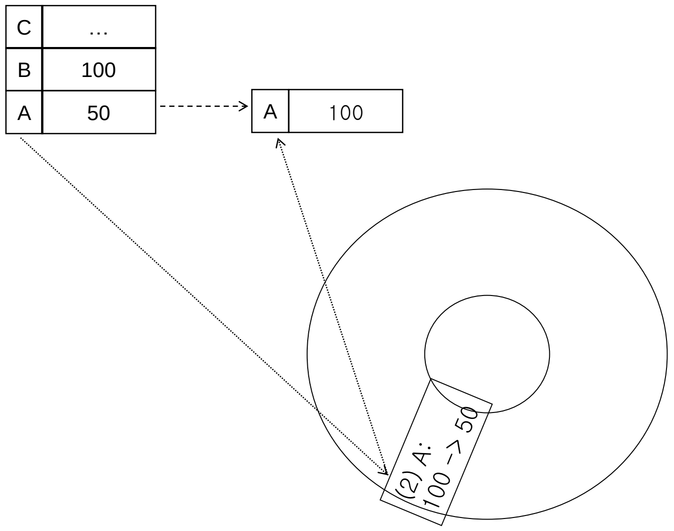


B) Multiversions and Oracle Read Consistency

Multi-version Read Consistency

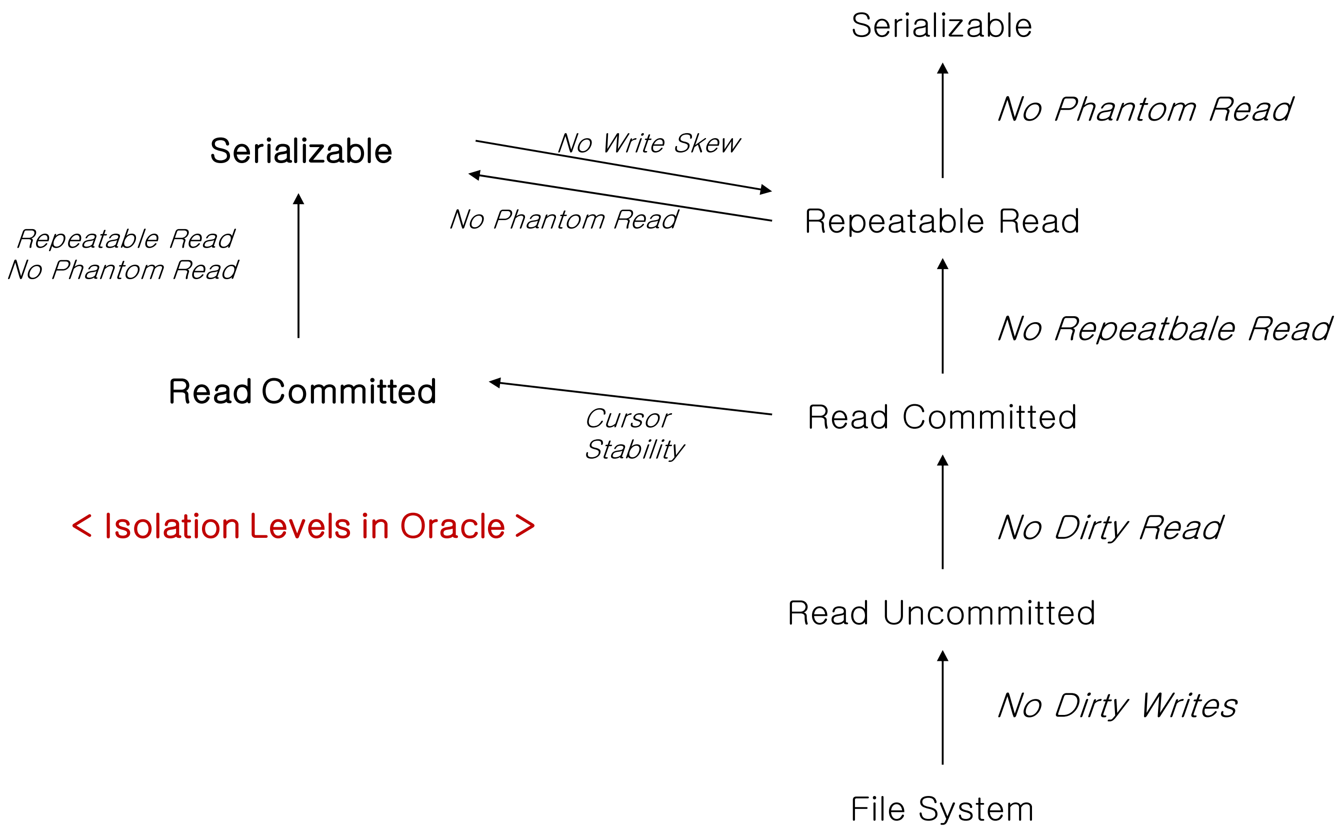
CC Mechanism in Oracle (5)

- Undo (or rollback) segment
  - For Multi-version Read Consistency in Oracle, we need to maintain the undo information. Undo(or rollback) segment is used for this purpose.
  - When the undo data is not available any more (e.g. due to space shortage in rollback segment), the "snapshot too old" error can be encountered



Undo (or Rollback) Segment
In Undo tablespace

CC Mechanism in Oracle(6): Isolation Levels in Oracle vs. ANSI Standard



< Isolation Levels in Oracle >

< Isolation Levels in ANSI Standard SQL >