Introduction to Database Systems IDBS - Spring 2024

Lecture 10 - Transactions in RDBMs

ACID Properties
Logging
Locking

Readings: PDBM 14

Omar Shahbaz Khan

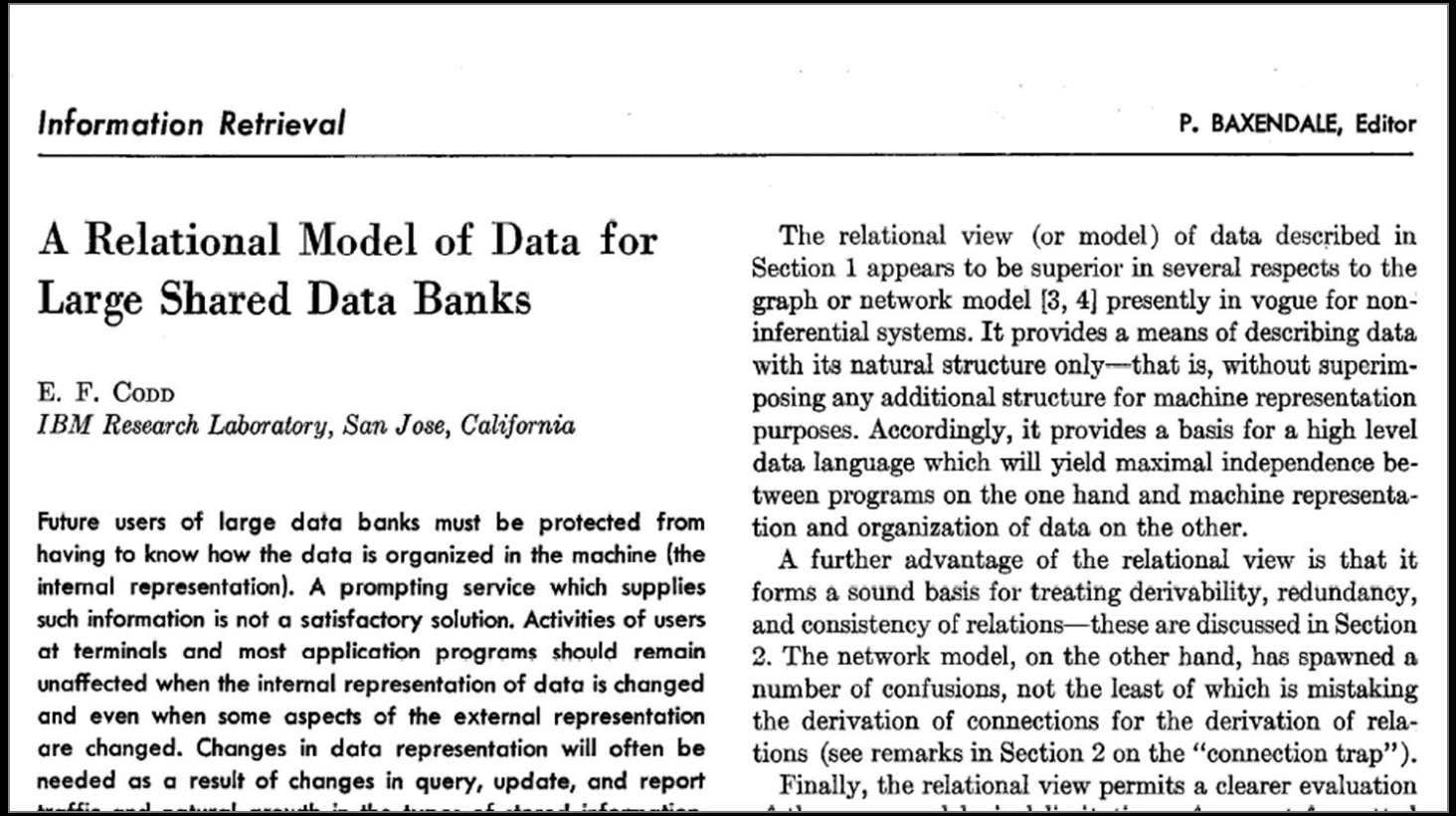
-- TODO

- DBMS History Lesson

- Transactions
  - ACID Properties
  - Buffer Management
  - Logging
  - Recovery
  - Locking

Story Time

The Beginning



Information Retrieval P. BAXENDALE, Editor

A Relational Model of Data for Large Shared Data Banks

E. F. CODD
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for non-inferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

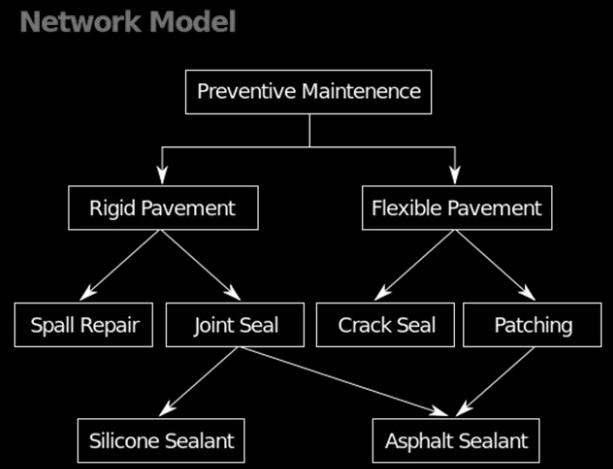
A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations—these are discussed in Section 2. The network model, on the other hand, has spawned a number of confusions, not the least of which is mistaking the derivation of connections for the derivation of relations (see remarks in Section 2 on the "connection trap").

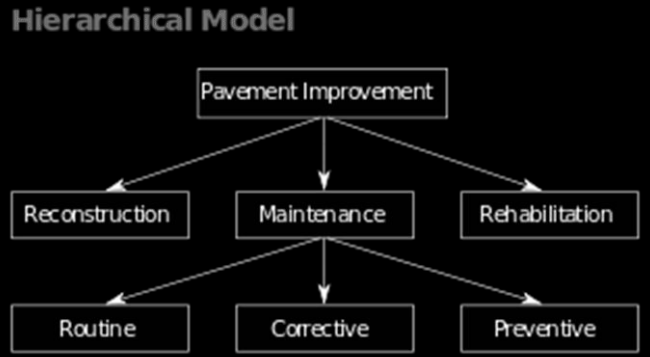
Finally, the relational view permits a clearer evaluation

Prehistoric Times

- Network model
  - CoDaSyL DBTG (GE)
  - Bachman, Turing award (73)
    “The Programmer as Navigator”

- Hierarchical model
  - IMS (IBM)





Problem with Existing Models

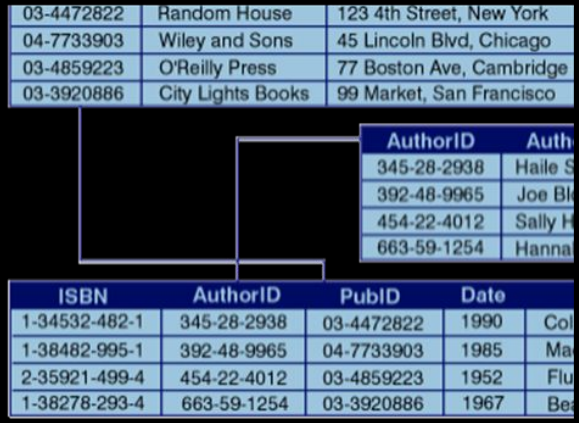
- Data and structure intertwined
  - “Queries” chase pointers
- Any change in structure invalidates existing “queries”

Now, consider the problem of printing out the part number, part name, and quantity committed for every part used in the project whose project name is “alpha.” The following observations may be made regardless of which available tree-oriented information system is selected to tackle this problem. If a program P is developed for this problem assuming one of the five structures above—that is, P makes no test to determine which structure is in effect—then P will fail on at least three of the remaining structures. More specifically, if P succeeds with structure 5, it will fail with all the others; if P succeeds with structure 3 or 4, it will fail with at least 1, 2, and 5; if P succeeds with 1 or 2, it will fail with at least 3, 4, and 5. The reason is simple in each case. In the absence of a test to determine which structure is in effect, P fails because an attempt is made to exceute a reference to a nonexistent file (available systems treat this as an error) or no attempt is made to execute a reference to a file containing needed information. The reader who is not convinced should develop sample programs for this simple problem.

Since, in general, it is not practical to develop application programs which test for all tree structurings permitted by the system, these programs fail when a change in structure becomes necessary.

Relational Model

- Simple mathematical model
- Natural declarative query languages
  - Calculus → SQL
  - Algebra → Internal representation



03-4472822	Random House	123 4th Street, New York
04-7733903	Wiley and Sons	45 Lincoln Blvd, Chicago
03-4859223	O'Reilly Press	77 Boston Ave, Cambridge
03-3920886	City Lights Books	99 Market, San Francisco

AuthorID	Auth
345-28-2938	Haile S
392-48-9965	Joe Bl
454-22-4012	Sally H
663-59-1254	Hanna

ISBN	AuthorID	PubID	Date	
1-34532-482-1	345-28-2938	03-4472822	1990	Col
1-38482-995-1	392-48-9965	04-7733903	1985	Ma
2-35921-499-4	454-22-4012	03-4859223	1952	Flu
1-38278-293-4	663-59-1254	03-3920886	1967	Be

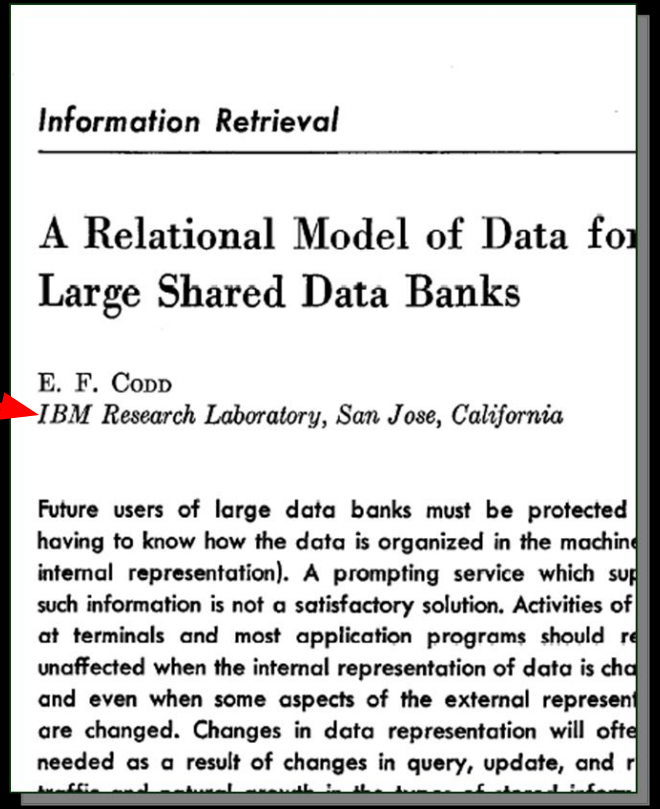
> This was as I say a revelation for me because Codd had a bunch of queries that were fairly complicated queries and since I'd been studying CODASYL, I could imagine how those queries would have been represented in CODASYL by programs that were five pages long that would navigate through this labyrinth of pointers and stuff. Codd would sort of write them down as one-liners. These would be queries like, "Find the employees who earn more than their managers." *[laughter]* He just whacked them out and you could sort of read them, and they weren't complicated at all, and I said, "Wow." This was kind of a conversion experience for me, that I understood what the relational thing was about after that.
>
> (Chamberlin, SQL Reunion)

Conflict?

- Hierarchical Model
  - IMS (IBM)

Within IBM, the trouble was the existing database product, IMS. The company had already invested, both financially and organizationally, in the infrastructure and expertise required to sell and support it. A radical new technology had a great deal to prove before it could displace a successful, reliable, revenue-generating product such as IMS. Initially, the threat was minimal; Codd published his original paper in the open literature because no one at IBM (himself included) recognized its eventual impact. The response to this publication from the outside technical community, however, soon showed the company that the idea had great commercial potential. To head off this eventuality, IBM quickly declared IMS its sole strategic product, thus setting up Codd and his work to be criticized as counter to company goals.

Funding a Revolution



Information Retrieval

A Relational Model of Data fo
Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected
having to know how the data is organized in the machin
internal representation). A prompting service which su
such information is not a satisfactory solution. Activities of
at terminals and most application programs should r
unaffected when the internal representation of data is cha
and even when some aspects of the external represen
are changed. Changes in data representation will ofte
needed as a result of changes in query, update, and r

The Next Years

- Debates: Comparison to older models
  - \+ Structure and simplicity of queries
  - ÷ Performance!? No “system”!

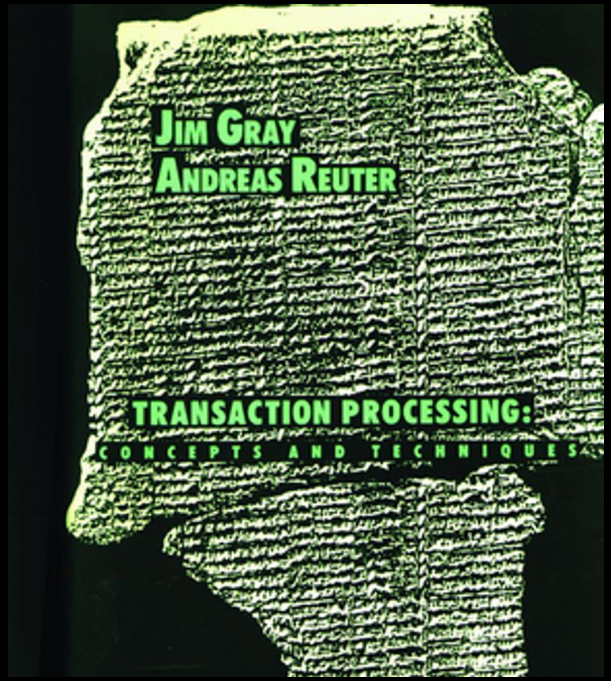
- Research projects
  - Ingres (Berkeley, 1973-1980+)
  - System R (IBM, 1974-1980+)

Major Milestones

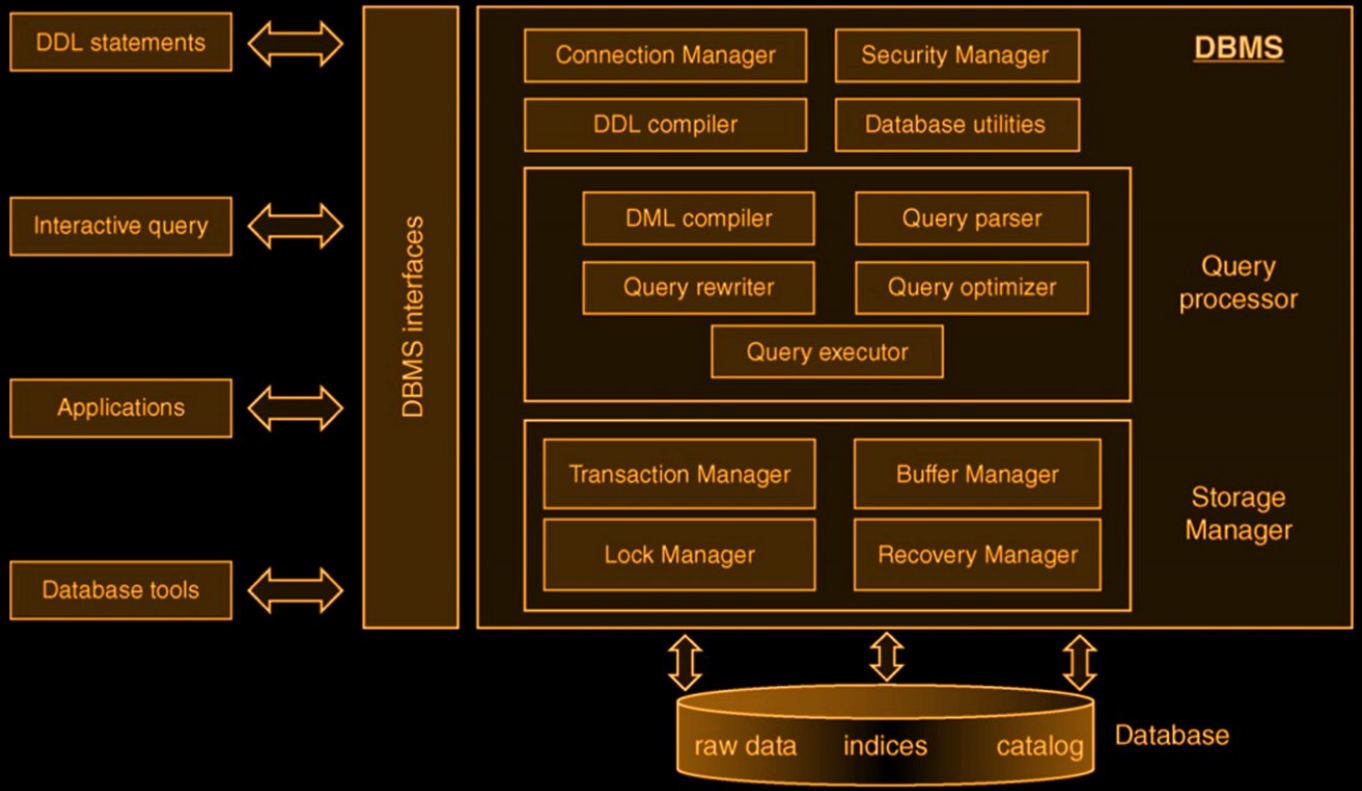
• Model (1970)	Codd, Turing award (81)
• SQL (1973-74)	Chamberlin
• Transactions (1975+)	Gray, Turing award (98)
• Logging (~1971)	
• Locking (1976)	Gray et al.
• B+ trees (1977)	Bayer, SIGMOD award
• Query optimization (1979)	Selinger et al.

• Hash join (1986)	Shapiro
• ARIES recovery (1989)	Mohan

DBMS Implementation: he “wrote the book”!



DBMS Architecture



-- TODO

✓ DBMS History Lesson

- Transactions
  - ACID Properties
  - Buffer Management
  - Logging
  - Recovery
  - Locking

Transactions

Why do we need transactions?

Consider the following transactions on the relation accounts(no, balance, type):

Transaction A

```
UPDATE accounts SET balance=balance*1.02 WHERE type='savings';
UPDATE accounts SET balance=balance*1.01 WHERE type='salary' AND balance > 0;
UPDATE accounts SET balance=balance*1.07 WHERE type='salary' AND balance < 0;
```

Transaction B

```
UPDATE accounts SET type='salary' WHERE no=12345;
```

Transaction C

```
UPDATE accounts SET balance=balance-1000 WHERE no=12345;
```

- Assume that account 12345 starts as 'savings' with balance=500.
- What are the possible balance values after running transactions A, B and C?

Why do we need transactions?

Consider the following transactions on the relation accounts(no, balance, type):

Transaction A

```
UPDATE accounts SET balance=balance*1.02 WHERE type='savings';
UPDATE accounts SET balance=balance*1.01 WHERE type='salary' AND balance > 0;
UPDATE accounts SET balance=balance*1.07 WHERE type='salary' AND balance < 0;
```

Transaction B

```
UPDATE accounts SET type='salary' WHERE no=12345;
```

Transaction C

```
UPDATE accounts SET balance=balance-1000 WHERE no=12345;
```

Assume that account 12345 starts as 'savings' with balance=500

A→B→C: -490 A→C→B: -490 B→A→C: -495

B→C→A: -535 C→A→B: -510 C→B→A: -535

Why do we need transactions?

Consider the following transactions on the relation accounts(no, balance, type):

Transaction A

```
UPDATE accounts SET balance=balance*1.02 WHERE type='savings';
UPDATE accounts SET balance=balance*1.01 WHERE type='salary' AND balance > 0;
UPDATE accounts SET balance=balance*1.07 WHERE type='salary' AND balance < 0;
```

Transaction B

```
UPDATE accounts SET type='salary' WHERE no=12345;
```

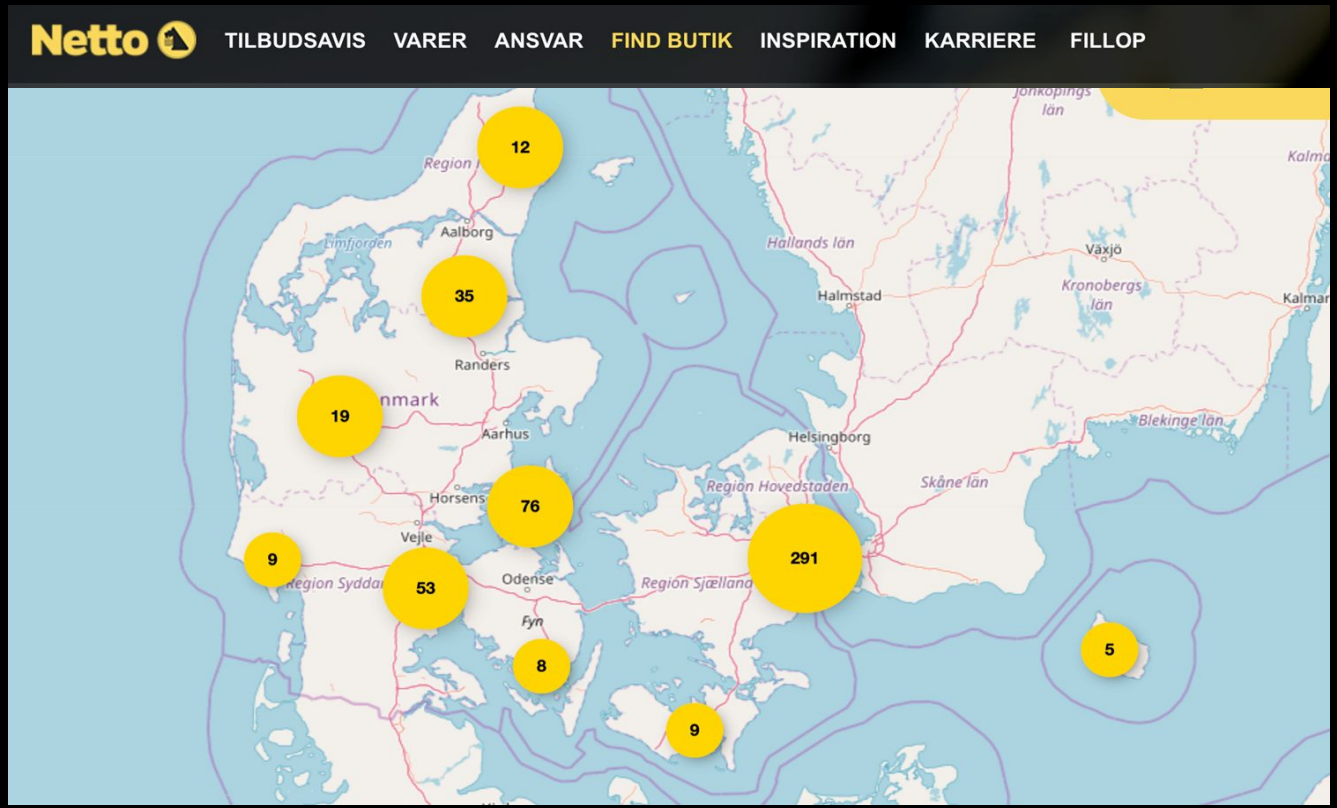
Transaction C

```
UPDATE accounts SET balance=balance-1000 WHERE no=12345;
```

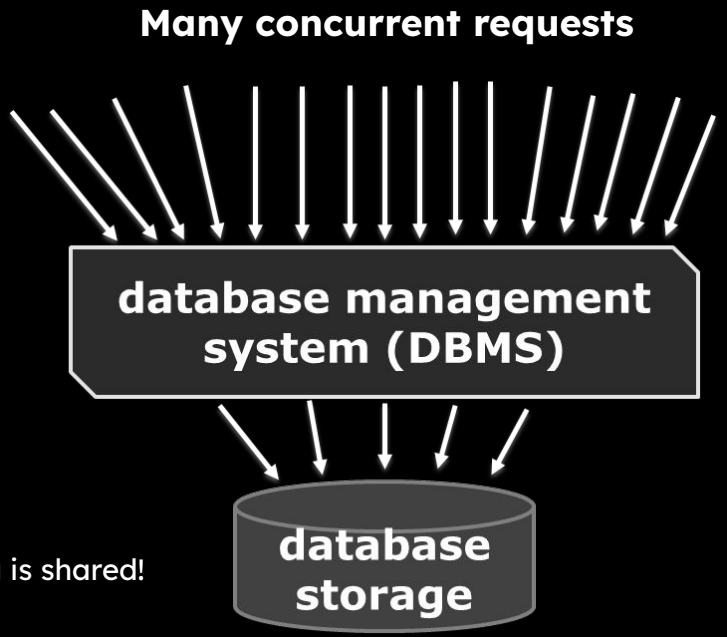
Assume that account 12345 starts as 'savings' with balance=500 and they are interleaving

A.1: 510
B: 510 (salary)
A.2: 515.10
C: -484.90
A.3: -518.843

Why do we need Transactions?



Why do we need Transactions?

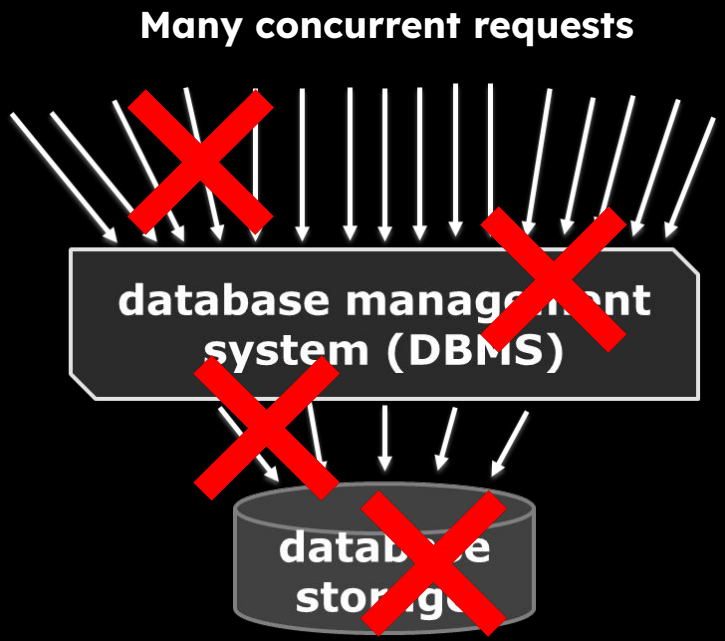


- from different applications
- from several users in the same application

DBMS must ensure reliable operations over shared data despite many concurrent accesses

Why do we need Transactions?

ANYTHING CAN CRASH!



DBMS must ensure reliable operations over shared data despite many concurrent accesses

To the end user everything is fine, thanks to transactions

Recovery	Constraints, Triggers	Concurrency Control	Recovery

ATOMICITY

A Transaction is “one operation”

CONSISTENCY

Each Transaction moves the DB from one consistent state to another

ISOLATION

Each Transaction is alone in the world

DURABILITY

Persistence of successful transactions even through system failure

ACID Properties of Transactions

- Atomicity: Each transaction runs to completion or has no effect at all.
- Consistency: After a transaction completes, the integrity constraints are satisfied.
- Isolation: Transactions executed in parallel have the same effect as if they were executed sequentially.
- Durability: The effect of a committed transaction remains in the database even if the system crashes.

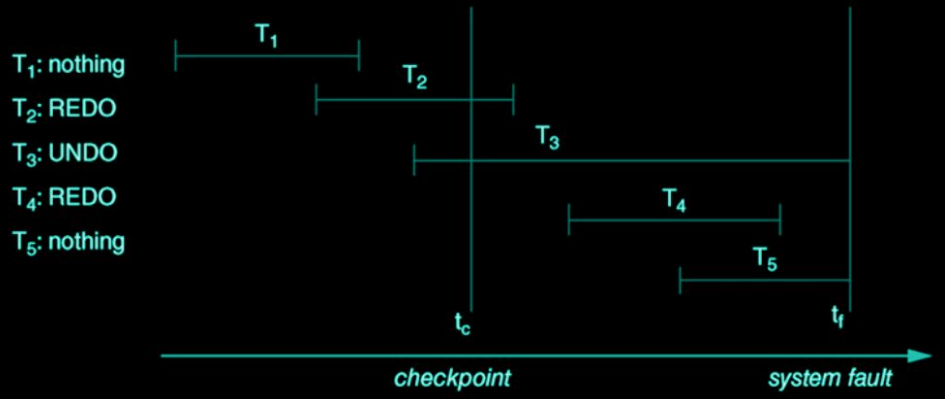
How to Implement Transactions?

- Consistency ~= satisfying constraints
  - Use indexes for primary and foreign key, triggers, …
- Atomicity and Durability = tracking changes
  - Logging “before” values to UNDO changes
  - Logging “after” values to REDO changes
- Isolation = preventing corrupting changes
  - Pessimistic: Locking to prevent conflicts
  - Optimistic: Time stamps to detect conflicts

Atomicity and Durability Issues

- Atomicity
  - Transactions abort
  - Systems crash
- Durability
  - Systems crash

- Upon restart
  - Want to see effects of T1, T2, T4
  - Want to remove the effects of T3, T5



Buffer Management (RAM v Disk)

- Once a transaction completes, it performs a COMMIT
- A COMMIT ≠ changes are on disk
- Buffer Management Policies determine when changes are moved from RAM to disk
  - The transaction log is always updated / written to disk
  - FORCE / NO FORCE policies affect data pages of committed transactions
  - STEAL / NO STEAL policies affect data pages of uncommitted transactions

FORCE / NO STEAL

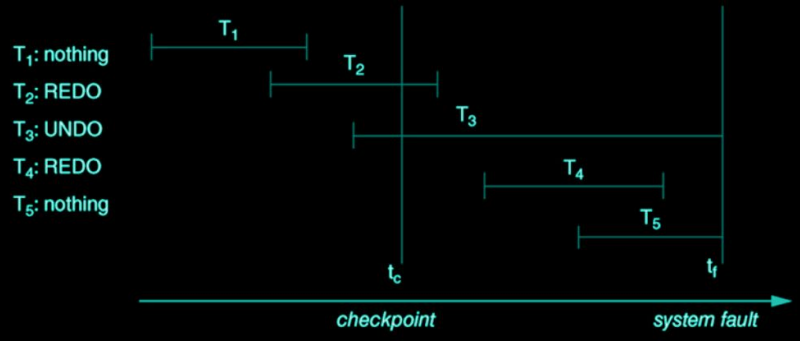
- FORCE = write changed pages to disk at COMMIT
  - Book: “immediate update” policy
  - Ensures Durability (assuming writes are atomic)
  - Increases response time
  - Will we FORCE changes to disk at COMMIT? NO

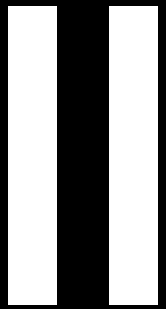
- NO STEAL = allow updated pages to be replaced
  - Book: NO STEAL = “deferred update” policy
  - Ensures Atomicity (can simply discard at abort)
  - Increases response time
  - Will we guarantee NO STEAL of dirty pages? NO

NO FORCE / STEAL

- NO FORCE: Changes in RAM after COMMIT
  - What if system crashes?
  - Need to remember the new value of P to be able to REDO the changes

- STEAL: Changes to disk before COMMIT
  - What if transaction aborts? system crashes?
  - Need to remember the old value of P to be able to UNDO the changes





Write-Ahead Logging (WAL Protocol)

- Write-Ahead Logging
  1. Before any changes are written to disk
     we force the corresponding log record to disk
  2. Before a transaction is committed
     we force all log records for the transaction to disk

- #1 ensures Atomicity
- #2 ensures Durability

Key Concept: The Log

- Write REDO and UNDO info to log
  - Ordered list of REDO/UNDO info
  - Think of an infinite file with append only!

- Log processing must be fast – why? and how?
  - Write minimal info to log (diff)
    - <xid, pageID, offset, length, old_data, new_data> + control info
    - Many log entries per page
  - Ensure sequential writes!
    - Writing a log entry ⇨ writing all previous entries
    - **Put log on its own disk!**

Transactions in MMDBMSs

- ACI…D
  - Are we ok with losing all data?
  - How do we make changes persistent?
    - Transaction Log on Disk
    - Snapshots on Disk (Copy of the database state)

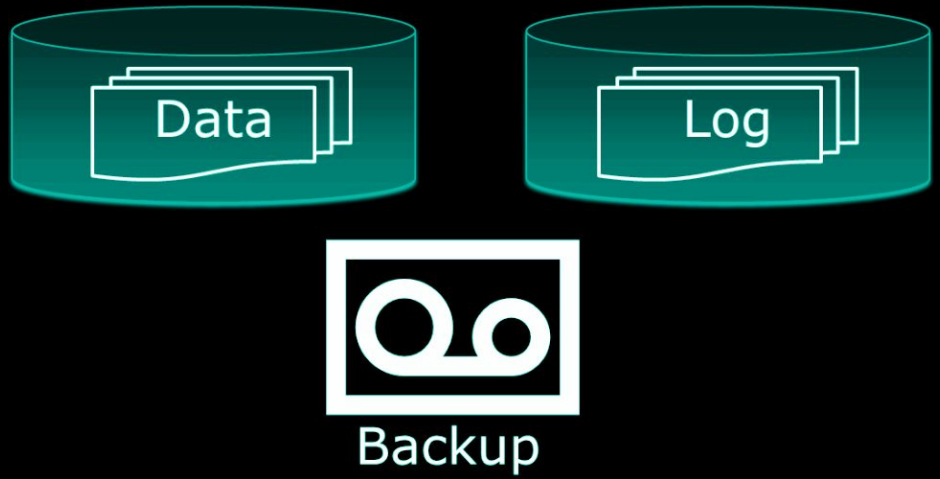
Restart Recovery

1. Analyze information about transactions from the last checkpoint

2. REDO the changes of committed transactions that did not make it to disk

3. UNDO the changes of uncommitted transactions that accidentally made it to disk

Database Setup for DBAs



- On crash:
  - Data + Log = Recovery succeeds
  - Backup + Log = Recovery succeeds
  - Data + Backup (No Log) = Recovery FAILS!

https://www.visualpharm.com/free-icons/set/hard-drive

High Availability

- Typical approach: Second (failover) server
  - Takes over in case the primary fails
- Transaction log used to keep it up to date

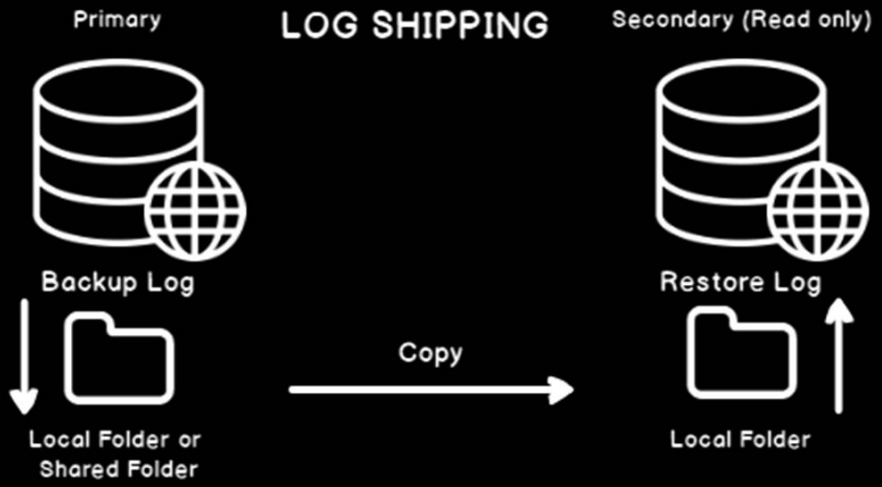


Image source: https://www.sqlshack.com/sql-server-transaction-log-and-high-availability-solutions/

How to Implement Transactions?

- Consistency ~= satisfying constraints
  - Use indexes for primary and foreign keys, triggers, …
- Atomicity and Durability = tracking changes
  - Logging “before” values to undo changes
  - Logging “after” values to redo changes
- Isolation = preventing corrupting changes
  - **Pessimistic: Locking to prevent conflicts**
  - Optimistic: Time stamps to detect conflicts

Isolation and Serializability

- Want transactions to satisfy *serializability*:
  - The state of the database should always look as if the committed transactions ran in some *serial schedule*

- The scheduler of the DBMS is allowed to choose the order of transactions:
  - It is not necessarily the transaction that is started first, which is first in the serial schedule

A Simple Scheduler

- A simple scheduler would maintain a queue of transactions and carry them out in order

- Problems:
  - Transactions must *wait* for each other, even if unrelated (e.g. requesting data on different disks)
    - Some transactions may take very long, e.g. when external input or remote data is needed during the transaction
  - Possibly smaller throughput (Why?)

A Simple Scheduler

- A simple scheduler would maintain a queue of transactions and carry them out in order
- Some believe this is fine for *transaction processing*, especially for MMDBs

The End of an Architectural Era
(It's Time for a Complete Rewrite)

Michael Stonebraker
Samuel Madden
Daniel J. Abadi
Stavros Harizopoulos

Nabil Hachem
AvantGarde Consulting, LLC
nhachem@agdba.com

Pat Helland
Microsoft Corporation
phelland@microsoft.com

Interleaving Schedulers

- Most DBMSs still have schedulers that allow the actions of transactions to interleave

- However, the result should be **as if** some serial schedule was used
  - A non-serial schedule that yields the same outcome as a serial schedule is known as a serializable schedule

- We will now study a mechanism that *enforces* “serializability”: **Locking**

- Other methods exist: Time stamping / optimistic concurrency control
  - Out of scope for this course

Locks

- In its simplest form, a lock is a right to perform operations on a database element
- Only one transaction may hold a lock on an element at any time
- Locks must be requested by transactions and granted by the locking scheduler
- Typically, two types of locks: Read/Shared or Write/Exclusive

Rigorous Two-Phase Locking

- Rigorous 2PL protocol:
  1. Before reading a record/page, get a shared (S) lock
     Before writing a record/page, get an exclusive (X) lock
  2. A record/page cannot have an X lock at the same time as any other lock
  3. Release all locks on COMMIT/ABORT

- R2PL is commonly implemented, since:
  - Simple to understand and works well in practice
  - It makes transaction rollback easier to implement

- But:
  Optimistic methods are growing in popularity!

Locks and Deadlocks

- The DBMS sometimes must make a transaction wait for another transaction to release a lock.
- This can lead to deadlock if e.g. A waits for B, and B waits for A.
- In general, we have a deadlock exactly when there is a cycle in the waits-for graph.
- Deadlocks are resolved by aborting some transaction involved in the cycle.

Avoiding Deadlocks

- Upgrade requests can also deadlock

SELECT :x=counter FROM table WHERE *<condition>* :x=:x+1 UPDATE table SET counter = :x WHERE *<condition>*	UPDATE table SET counter = counter+1 WHERE *<condition>* SELECT :x=counter FROM table WHERE *<condition>*	SELECT FOR UPDATE :x=counter FROM table WHERE *<condition>* :x=:x+1 UPDATE table SET counter = :x WHERE *<condition>*

- Order Matters:
  - With consistent order of access, deadlocks are avoided
  - Why do B+ tree accesses not deadlock? - Same traversal order
  - Optimizer may not allow control over order!

Phantom Tuples

- Suppose we lock tuples where A=1 in a relation, and subsequently another tuple with A=1 is inserted.
- For some transactions this may result in unserializable behaviour, i.e., it will be clear that the tuple was inserted during the course of a transaction.
- Such tuples are called phantoms.

Phantom Example

```
CREATE TABLE B (x INTEGER, y INTEGER);
INSERT INTO B VALUES (1, 2);


BEGIN;
SELECT MIN(y) FROM B WHERE x = 1;
```

```
BEGIN;
INSERT INTO B VALUES (1, 1);
COMMIT;
```

```
-- Repeat the SAME read!
SELECT MIN(y) FROM B WHERE x = 1;
COMMIT;
```

Avoiding Phantoms

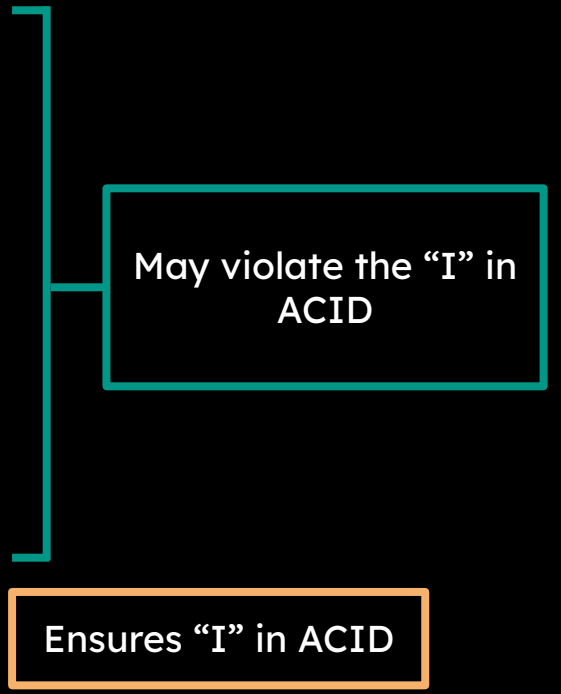
- Phantoms can be avoided by putting an exclusive lock on a relation before adding tuples.
  - However, this leads to poor concurrency.

- A technique called “index locking” can be used to prevent other transactions from inserting phantom tuples but allow most non-phantom insertions.

- In SQL, the programmer may choose to either allow phantoms in a transaction or insist they should not occur.

Isolation Levels in Modern Systems

- **READ UNCOMMITTED**
  a transaction can read uncommitted changes
- **READ COMMITTED**
  a transaction only reads committed data,
  some other transaction may overwrite this data
- **REPEATABLE READ**
  a transaction only reads committed data,
  other transactions cannot overwrite this data,
  but phantoms are possible
- **SERIALIZABLE**
  ensures serializable schedule with no anomalies



Database	Default Isolation	Maximum Isolation
Actian Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	?
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	RR
MySQL 5.6	RR	S
MemSQL 1b	RC	RC
MS SQL Server 2012	RC	S
NuoDB	CR	CR
Oracle 11g	RC	SI
Oracle Berkeley DB	S	S
Oracle Berkeley DB JE	RR	S
Postgres 9.2.2	RC	S
SAP HANA	RC	SI
ScaleDB 1.02	RC	RC
VoltDB	S	S
Legend	*RC: read committed, RR: repeatable read, S: serializability, SI: snapshot isolation, CS: cursor stability, CR: consistent read*	

The entire world doesn't run on ACID!

But an important part of it does so!

Figure taken from: [When is "ACID" ACID? Rarely]

-- TODO -> DONE

- ✓ DBMS History Lesson
- ✓ Transactions
  - ✓ ACID Properties
  - ✓ Buffer Management
  - ✓ Logging
  - ✓ Recovery
  - ✓ Locking

Takeaways

You're banished to
the Phantom Zone

RDBMS Implementation:

- Queries: SQL and Query Optimization
- Transactions: Locking and Logging
- Based on old assumptions

Next Time in IDBS...

Introduction to Database Systems IDBS - Spring 2024

Lecture 11 - Scale and Analytics

Scaling-Out
NoSQL
Eventual Consistency
CAP Theorem

Readings: PDBM 11

Omar Shahbaz Khan