Database System Architecture

Instructor: Matei Zaharia
Outline

System R discussion

Relational DBMS architecture

Alternative architectures & tradeoffs
Outline

System R discussion

Relational DBMS architecture

Alternative architectures & tradeoffs
System R Design

Already had essentially the same architecture as a modern RDBMS!

» SQL
» Many storage & access methods (B-trees, etc)
» Cost-based optimizer
» Compiling queries to assembly
» Lock manager
» Recovery via log + shadow pages
» View-based access control
System R Motivation

Navigational DBMS are hard to use

Can relational DBMS really be practical?
Why is the relational model more flexible?
Three Phases of Development

Why was System R built in 3 phases?
Storage in System R Phase 0

What was the issue with this design?

Too many I/Os:
• For each tuple, look up all its fields
• Use “inversions” to find TIDs with a given value for a field

Can also have reverse mappings (inversions)
Storage in System R Phase 1

B-tree nodes contain values of the column(s) indexed on.

Data pages can contain all fields of the record.

Give an example query that would be faster with B-Trees!
API

Mostly the same SQL language as today

Embedded SQL in PL/I and COBOL
  » .NET added LINQ in 2007

Interesting additions:
  » “EXISTS”
  » “LIKE”
  » Prepared statements
  » Outer joins

```
SELECT expression(s)
FROM table
WHERE EXISTS
  (SELECT expr FROM table WHERE cond)
WHERE name LIKE 'Mat%'
stmt = prepare("SELECT name FROM table WHERE id=?")
execute(stmt, 5)
```
Query Optimizer

How did the System R optimizer change after Phase 0?
Query Compilation

Why did System R compile queries to assembly code?

How did it compile them?

Do databases still do that today?
Example 1:

```sql
SELECT SUPPNO, PRICE
FROM QUOTES
WHERE PARTNO = '010002'
AND MINQ <= 1000 AND MAXQ >= 1000;
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>CPU time (msec on 168)</th>
<th>Number of I/Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parsing</td>
<td>13.3</td>
<td>0</td>
</tr>
<tr>
<td>Access Path Selection</td>
<td>40.0</td>
<td>9</td>
</tr>
<tr>
<td>Code Generation</td>
<td>10.1</td>
<td>0</td>
</tr>
<tr>
<td>Fetch answer set</td>
<td>1.5</td>
<td>0.7</td>
</tr>
</tbody>
</table>
Recovery

**Goal:** get the database into a consistent state after a failure

“A consistent state is defined as one in which the database does not reflect any updates made by transactions which did not complete successfully.”
Recovery

Three main types of failures:
» Disk (storage media) failure
» System crash
» Transaction failure
Handling Storage Failure

Main disk

DBMS

Tables

Change log

Backup disk

(Older) tables

RAM

Change log

Backup

tables
System Crash Failure

DBMS

RAM

Buffered pages, in-progress transactions

Tables

Change log

Change log

(Older) tables

Main disk

Backup disk

Table 1: Main Disk Tables

<table>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Smith</td>
<td>123 Main St</td>
</tr>
<tr>
<td>Jane</td>
<td>Doe</td>
<td>456 Front St</td>
</tr>
<tr>
<td>Mike</td>
<td>Miller</td>
<td>789 Back St</td>
</tr>
</tbody>
</table>

Table 2: Change Log

<table>
<thead>
<tr>
<th>Item</th>
<th>Action</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Add</td>
<td>1.0</td>
</tr>
<tr>
<td>Age</td>
<td>Update</td>
<td>2.0</td>
</tr>
<tr>
<td>Addr</td>
<td>Delete</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Table 3: Change Log (Older)

<table>
<thead>
<tr>
<th>Item</th>
<th>Action</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Add</td>
<td>4.0</td>
</tr>
<tr>
<td>Age</td>
<td>Update</td>
<td>5.0</td>
</tr>
<tr>
<td>Addr</td>
<td>Delete</td>
<td>6.0</td>
</tr>
</tbody>
</table>
Handling Crash Failures: Shadow Pages

Why do we need both shadow pages and a change log?
A Later Note on Recovery

In retrospect, we regret not supporting the LOG and NO SHADOW option. As explained in Section 3.8, the log makes shadows redundant, and the shadow mechanism is quite expensive for large files.

Transaction Failure

BEGIN TRANSACTION;

SELECT balance FROM accounts
WHERE user_id = 1;

UPDATE accounts WHERE user_id = 1
SET balance = balance - 100;
COMMIT TRANSACTION;

ROLLBACK TRANSACTION;
Handling Transaction Failures

Just undo any changes they made, which we logged in the change log

Nobody else “saw” these changes due to System R’s locking mechanism
Locking

The problem:
» Different transactions are concurrently trying to read and update various data records
» Each transaction wants to see a static view of the database (maybe lock whole DB)
» For efficiency, we can’t let them do that!
Fundamental Tradeoff

Finer-grained locking

- Lock smaller units of data (records or fields), lock for specific operations (e.g. R/W)
  - Allows more transactions to run concurrently
  - More runtime overhead

Coarser-grained locking

- Lock bigger units of data (e.g. whole table) for broader purposes (e.g. all operations)
  - More efficient to implement
  - Less concurrency

Even if fine-grained locking were free, in some cases where it would give unacceptable perf!
Fundamental Tradeoff

- **Finer-grained locking**
  - Lock smaller units of data (records or fields), lock for specific operations (e.g. R/W)
  - + Allows more transactions to run concurrently
  - − More runtime overhead

- **Coarser-grained locking**
  - Lock bigger units of data (e.g. whole table) for broader purposes (e.g. all operations)
  - + More efficient to implement
  - − Less concurrency

- **Strong isolation level**
  - Closer to exclusive view of DB (can’t see others’ changes)

- **Weak isolation level**
  - See others’ changes, but more concurrency
Locking and Isolation in System R

Locking:
» Started with “predicate locks” based on expressions: too expensive
» Moved to hierarchical locks: record/page/table, with read/write types and intentions

Isolation levels:
» Level 1: Transaction may read uncommitted data; successive reads to a record may return different values
» Level 2: Transaction may only read committed data, but successive reads can differ
» Level 3: Successive reads return same value

Most apps chose Level 3 since others weren’t much faster
Are There Alternatives to Locking for Concurrency?
Authorization

**Goal:** give some users access to just parts of the database

» A manager can only see and update salaries of her employees
» Analysts can see user IDs but not names
» US users can’t see data in Europe
Authorization

System R used view-based access control

» Define SQL views (queries) for what the user can see and grant access on those

```sql
CREATE VIEW canadian_customers AS
SELECT customer_name, email_address
FROM customers
WHERE country = "Canada";
```

Elegant implementation: add the user’s SQL query on top of the view’s SQL query
User Evaluation

How did the developers evaluate System R?

What was the user feedback?
Outline

System R discussion

Relational DBMS architecture

Alternative architectures & tradeoffs
Typical RDBMS Architecture

- Query Planner
- Query Parser
- User Transaction
- Transaction Manager
- Concurrency Control
- Lock Table
- Buffer Manager
- Recovery Manager
- File Manager
- Mem.Mgr. Buffers
- Log
- Indexes
- Data Statistics
- User Data
- System Data

User
Boundaries

Some of the components have clear boundaries and interfaces for modularity
  » SQL language
  » Query plan representation (relational algebra)
  » Pages and buffers

Other components can interact closely
  » Recovery + buffers + files + indexes
  » Transactions + indexes & other data structures
  » Data statistics + query optimizer
Differentiating by Workload

2 big classes of commercial RDBMS today

**Transactional DBMS**: focus on concurrent, small, low-latency transactions (e.g. MySQL, Postgres, Oracle, DB2) → real-time apps

**Analytical DBMS**: focus on large, parallel but mostly read-only analytics (e.g. Teradata, Redshift, Vertica) → “data warehouses”
How To Design Components for Transactional vs Analytical DBMS?

<table>
<thead>
<tr>
<th>Component</th>
<th>Transactional DBMS</th>
<th>Analytical DBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data storage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Locking</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recovery</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# How To Design Components for Transactional vs Analytical DBMS?

<table>
<thead>
<tr>
<th>Component</th>
<th>Transactional DBMS</th>
<th>Analytical DBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data storage</td>
<td>B-trees, row oriented storage</td>
<td>Column-oriented storage</td>
</tr>
<tr>
<td>Locking</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recovery</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# How To Design Components for Transactional vs Analytical DBMS?

<table>
<thead>
<tr>
<th>Component</th>
<th>Transactional DBMS</th>
<th>Analytical DBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data storage</td>
<td>B-trees, row oriented storage</td>
<td>Column-oriented storage</td>
</tr>
<tr>
<td>Locking</td>
<td>Fine-grained, very optimized</td>
<td>Coarse-grained (few writes)</td>
</tr>
<tr>
<td>Recovery</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# How To Design Components for Transactional vs Analytical DBMS?

<table>
<thead>
<tr>
<th>Component</th>
<th>Transactional DBMS</th>
<th>Analytical DBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data storage</td>
<td>B-trees, row oriented storage</td>
<td>Column-oriented storage</td>
</tr>
<tr>
<td>Locking</td>
<td>Fine-grained, very optimized</td>
<td>Coarse-grained (few writes)</td>
</tr>
<tr>
<td>Recovery</td>
<td>Log data writes, minimize latency</td>
<td>Log queries</td>
</tr>
</tbody>
</table>
Outline

System R discussion

Relational DBMS architecture

Alternative architectures & tradeoffs
How Can We Change the DBMS Architecture?
Decouple Query Processing from Storage Management

Example: data lake systems (Hadoop, GFS, Athena)

Processing engines:
- MapReduce
- Spark
- Apache Storm
- TensorFlow
- Presto

Open storage and metadata formats:
- Parquet
- JSON
- Delta Lake
- Hive

Large-scale file systems or blob stores:
- GFS
- HDFS
- Amazon S3
Decouple Query Processing from Storage Management

Pros:
» Can scale compute independently of storage (e.g. in public cloud)
» Let different orgs develop different engines
» Your data is “open” by default to new tech

Cons:
» Harder to guarantee isolation, reliability, etc
» Harder to co-optimize compute and storage
» Can’t optimize across multiple engines
» Harder to manage if too many engines!
Change the Data Model

Key-value stores: data is just key-value pairs, don’t worry about record internals

Message queues: data is only accessed in a specific FIFO order; limited operations

ML frameworks: data is tensors, models, etc
Change the Compute Model

Stream processing: Apps run continuously and system can manage upgrades, scale-up, recovery, etc

Eventual consistency: handle it at app level
Different Hardware Setting

**Distributed databases:** need to distribute your lock manager, storage manager, etc, or find system designs that eliminate them.

**Public cloud:** “serverless” databases that can scale compute independently of storage (e.g. AWS Aurora, Google BigQuery)
AWS Aurora Serverless
Summary

All data systems face similar issues: API, performance, reliability, concurrency, etc.

Relational DBMS offer one architecture that tackles many of these concerns together.

One trend is to **break apart** this monolithic architecture into specialized components.