Table of Content

- Storage Hierarchy
- 5-Layer Architecture
- Overview: Layer-by-Layer
## 2010: Price versus speed

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Access Time</th>
<th>Cost per GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>1-10 ns/byte</td>
<td>-</td>
</tr>
<tr>
<td>Cache</td>
<td>10-60 ns/cache line</td>
<td>-</td>
</tr>
<tr>
<td>Main Memory</td>
<td>100-300 ns/block</td>
<td>~200 €/GB</td>
</tr>
<tr>
<td>Disk</td>
<td>10-20 ms/block</td>
<td>~1 €/GB</td>
</tr>
<tr>
<td>Tape</td>
<td>Difference ~10^4</td>
<td>&lt; 1 €/GB</td>
</tr>
</tbody>
</table>

**Difference**

- Really expensive: ~10^5
- Very expensive: ~10^4
2010: Storage Hierarchy

<table>
<thead>
<tr>
<th>Really expensive</th>
<th>Register</th>
<th>1-4 byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very expensive</td>
<td>Cache</td>
<td>1-4 MB</td>
</tr>
<tr>
<td>~ 200 € / GB</td>
<td>Main Memory</td>
<td>1-16 GB</td>
</tr>
<tr>
<td>~ 1 € / GB</td>
<td>Disk</td>
<td>512GB – 1TB discs</td>
</tr>
<tr>
<td>&lt; 1€/GB</td>
<td>Tape</td>
<td>“Infinite” tape robots</td>
</tr>
</tbody>
</table>
### 2016: Storage Hierarchy

<table>
<thead>
<tr>
<th>Level</th>
<th>Storage Type</th>
<th>Cost per GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Really expensive</td>
<td>Register</td>
<td>1 – 32 byte</td>
</tr>
<tr>
<td>Very expensive</td>
<td>Cache</td>
<td>1-16 MB</td>
</tr>
<tr>
<td>~ 7 € / GB</td>
<td>Main Memory</td>
<td>16-256 GB</td>
</tr>
<tr>
<td>~ 0,04 € / GB</td>
<td>Disk</td>
<td>1-16 TB</td>
</tr>
<tr>
<td>~ “Infinite” tape</td>
<td>Tape</td>
<td>“Infinite” tape robots</td>
</tr>
</tbody>
</table>

![Storage Hierarchy Diagram]
Costs Drop Faster than you Think

Source: http://analystfundamentals.com/?p=88
New Players

- Really expensive Register: 1-10ns / byte
- Very expensive Cache: 10-100ns / cache line
- Main Memory: 60-300ns / block
- Solid-State Disks (SSD): 1 ms / block
- Disk: 10-20 ms / block
- Tape: sec – min

~ 7 € / GB
~ 1 € / GB
~ 0.04 € / GB
New Players

https://www.pcworld.com/article/3011441/
Characteristics

random access $\neq$ sequential

Quelle: http://blog.laptopmag.com/faster-than-an-ssd-how-to-turn-extra-memory-into-a-ram-disk
Prize of Main Memory

- 2014: 1TB DRAM ~ 5000€
- 2016: Laptops with 16GB, desktops with 32GB, servers with 128GB
- 2019: Mobiles with 32GB, servers with >1TB
- My Guess: 99% of all commercial databases are smaller than 100GB
  - Research: Main memory databases
Consequences

• Dealing with memory hierarchy is core concern of DBMS
  – Another issue is multi-core
• This lecture will mostly focus on disk versus RAM
• Similar problems for cache-RAM, disk-SSD, ...
• Many differences between storage media
  – Speed, durability, size, cost
  – Block sizes
  – Read/write, random-access/sequential
  – Error rates, longevity
  – ...

Table of Content

- Storage Hierarchy
- 5-Layer Architecture
- Overview: Layer-by-Layer
Five Layer Architecture

Conceptual
- Data Model
  - Schema, SQL, data types

Logical
- Logical Access
  - Records, transactions
- Data Structures
  - Arrays, locks

Physical
- Buffer Management
  - Memory blocks (pages)
- Operating System
  - Disks, disc blocks
Tasks

- Data Model
  - Logical Access
    - Data Structures
      - Buffer Management
        - Operating System
          - Query optimization
          - Access control
          - Integrity constraints
          - Physical record manager
          - Index manager
          - Lock manager
          - Log / Recovery

- Transaction processing
  - Cursor management
- Block management
  - Caching
- Sort
Operations

SQL: select ... from ... Where
Grant access to ...
Create index on ...

OPEN – FETCH –CLOSE
STORE Record

READ page
WRITE page

Data Model

Logical Access

Data Structures

Buffer Management

Operating System

RECORDs in pages
access paths, indexes
Note: Idealized Representation

- Layers **may be merged**
  - E.g. logical and internal record-based layers
- Not all functionality can be assigned to exactly one layer
  - E.g. recovery, optimization
- Layers sometimes must **access non-neighboring layers**
  - Prefetching needs to know the query
    - Layer 4 to Layer 1/2
  - Optimizer needs to know about physical data layout
    - Layer 1 to layer 4/5
  - Breaks **information hiding** principle
Table of Content

- Storage Hierarchy
- 5-Layer Architecture
- Overview: Layer-by-Layer
Bottom-Up

- Data Model
- Logical Access
- Data Structures
- Buffer Management
- Operating System
Classical Discs

- **Durable**, slow, cheap, large, robust (compared to ...)
- In principle: *Same read/write speed*
- Much difference between random-access / scan
• Redundancy: **Fail-safety and access speed**
  – Increased read performance, write perf. not affected (parallel write)
  – Disc crash (one) can be tolerated
  – Be careful about dependent components (controller, power, ...)

• **Drawbacks**
  – Which value is correct in *case of divergence* in the two copies?
  – Space consumption doubles
Bottom-Up

- Data Model
- Logical Access
- Data Structures
- Buffer Management
- Operating System

Records, Blocks, Files
Access Methods: Sequential Unsorted Files

- Access to records by record/tuple identifier (RID or TID)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1522</td>
<td>Bond</td>
<td>...</td>
</tr>
<tr>
<td>123</td>
<td>Mason</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1754</td>
<td>Miller</td>
<td>...</td>
</tr>
</tbody>
</table>

- Operations
  - **INSERT( Record):** Move to end of file and add, O(1)
  - **SEEK( TID):** Sequential scan, O(n)
    - FIRST ( File): O(1)
    - NEXT( File): O(1)
    - EOF ( File): O(1)
  - **DELETE( TID):** Seek TID; flag as deleted, O(n)
  - **REPLACE( TID, Record):** Seek TID; write record, O(n)
    - What happens if records have variable size?
Access Methods: Sequential sorted Files

<table>
<thead>
<tr>
<th>TID</th>
<th>Name</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Mason</td>
<td>...</td>
</tr>
<tr>
<td>1522</td>
<td>Bond</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1754</td>
<td>Miller</td>
<td>...</td>
</tr>
</tbody>
</table>

- **Operations**
  - **SEEK(TID):** Bin search, $O(\log(n))$
    - But a lot of random access
    - Might be slower than scanning the file
  - **INSERT(Record):** Seek(TID), move records by one, $O(n)$
    - This is terribly expensive
  - ...
Indexed Files

- Operations
  - SEEK( TID): Using order in TIDs: $O(\log(n))$
    - Only if tree is balanced
    - Only if tree is ordered by the right attribute
  - INSERT( TID): Seek TID and insert; possibly restructuring
  - ...

![Indexed File Diagram]

- Root
- Internal Node
- Leaves

Indexed Files

- Operations
  - SEEK( TID): Using order in TIDs: $O(\log(n))$
    - Only if tree is balanced
    - Only if tree is ordered by the right attribute
  - INSERT( TID): Seek TID and insert; possibly restructuring
  - ...

![Indexed File Diagram]

- Root
- Internal Node
- Leaves
Storage in Oracle

- Data files are assigned to **tablespaces**
  - May consist of multiple files
  - All data from one object (table, index) are in one tablespace
    - But table and index can be in different ones
  - Backup, quotas, access, ...

- Extents: **Continuous sequences** of blocks on disc

- Space is allocated in extents (min, next, max, ...)

- Segments logically group all extents of an object
Managing space in Oracle

Tablespace (gestrichelter Bereich)

Datenbankdateien  Objekte (Segmente)
Bottom-Up

- Data Model
- Logical Access
- Data Structures
- Buffer Management
- Operating System

Virtual – physical blocks, access paths
Caching = Buffer Management

- Which blocks should be cached – for how long?
- Caching data blocks? Index blocks?
- **Competition**: Intermediate data, data buffers, sort buffer, ...
From Buffers to Records

- Absolute addressing: TID = <PageId, Offset, ID>
  - Pro: Fast access
  - Con: Records cannot be moved

- Absolute addressing + search: TID = <PageId,ID>
  - Pro: Records can be moved within page
  - Con: Slower access
Free Space, TX, and Concurrent Processes

- Oracle procedure for finding free space
- Free space managed at the level of segments
  - Logical database objects
- Explanation
  - TFL: transaction free list
  - PFL: process free list
  - MFL: master free list
  - HWM: High water mark

Database
Tablespace
Segment
Extent
OracleBlock
Bottom-Up

- Data Model
- Logical Access
- Data Structures
- Buffer Management
- Operating System

Query optimization
The ANSI/SPARC Three Layer-Model

- Query rewriting, view expansion
- Query execution plan generation and optimization: Access paths, join order, ...
- Execution of operators, pipelining
Query Processing

- **Declarative query**
  
  ```
  SELECT Name, Address, Checking, Balance  
  FROM   customer C, account A  
  WHERE Name = "Bond" and C.Account# = A.Account#
  ```

- **Translated in procedural Query Execution Plan (QEP)**
  
  ```
  FOR EACH c in CUSTOMER DO  
    IF c.Name = "Bond" THEN  
      FOR EACH a IN ACCOUNT DO  
        IF a.Account# = c.Account# THEN  
  ```
One Query – Many QEPs

FOR EACH c in CUSTOMER DO
  IF c.Name = “Bond” THEN
    FOR EACH a IN ACCOUNT DO

FOR EACH a in ACCOUNT DO
  FOR EACH c IN CUSTOMER DO
    IF a.Acco# = c.Acco# THEN

FOR EACH c in CUSTOMER WITH Name=“Bond” BY INDEX DO
  FOR EACH a IN ACCOUNT DO

FOR EACH c in CUSTOMER WITH Name=“Bond” BY INDEX DO
  FOR EACH a IN ACCOUNT with a.Acco#=c.Acco# BY INDEX DO

...
Query optimization

- **Task**: Find the (hopefully) **fastest QEP**
- **Two interdependent** levels: Best plan, best implementation
  - Different QEPs by **algebraic rewriting**
    - P1: $\sigma_{\text{Name}=\text{Bond}}(\text{Account} \bowtie \text{Customer})$
    - P2: $\text{Account} \bowtie \sigma_{\text{Name}=\text{Bond}}(\text{Customer})$
  - Different QEPs by **different operator implementations**
    - P1': Access by scan, hash-join
    - P1'': Access by index, nested-loop-join
- **Plan space**: Enumerate and evaluate (some? all?) QEPs
- **Optimization goal**: Minimize **size of intermediate results**
  - Might miss optimality in terms of runtime
    - Expansive subplan with sorted result
    - Cheap subplan with unsorted result
Cost-Based Optimizer

• Use **statistics on** current state of relations
  – Size, value distribution, fragmentation, cluster factors, ...

  ```sql
  FOR EACH a in ACCOUNT DO
    FOR EACH c IN CUSTOMER DO
      IF a.Account# = c.Account# THEN
        IF c.Name = "BOND" THEN ...
  ```

  – Let selectivity of $\sigma_{\text{Name}=\text{Bond}}$ be 1%, $|\text{Customer}|=10.000$, $|\text{Account}|=12.000$, Customer/Account evenly distributed
  – Performs ...
    • Join: $10.000 \times 12.000 = 120\text{M comparisons}$
    • Produces $\sim 12.000$ intermediate result tuples
    • Filters down to $\sim 120$ results
Join methods

- Suppose the previous query would contain no selection
- Can't we do better than “Join: 120M comparisons”
- Join methods
  - Nested loop join: $O(m*n)$ key comparisons
  - Sort-merge join
    - First sort relations in $O(n \log(n) + m \log(m))$
    - Merge results in $O(m+n)$
    - Sometimes better, sometimes worse
  - Hash join, index-join, grace-join, zig-zag join, ...
- Note: Complexity here measures number of comparisons
  - This is a “main-memory” viewpoint
  - Must not be used for IO tasks
Transactions (TX)

- Transaction: “Logical unit of work”
  ```
  Begin_Transaction
  UPDATE ACCOUNT
  SET Savings = Savings + 1M
  SET Checking = Checking - 1M
  WHERE Account# = 007;
  INSERT JOURNAL <007, NNN, "Transfer", ...>
  End_Transaction
  ```

- ACID properties
  - Atomic execution
  - Consistent DB state after commits
  - Isolation: No influence on result by concurrent TX
  - Durability: After commit, changes are reflected in the database
Lost Update Problem

Deposit $1,000
Read account value

5,000

Add $1,000
6,000

Write back

Deposit $2,000
Read account value

5,000

Add $2,000
7,000

Write back
Synchronization and schedules

\[ T_1: \begin{align*}
&\text{read } A; \\
&\text{A := } A - 10; \\
&\text{write } A; \\
&\text{read } B; \\
&\text{B := } B + 10; \\
&\text{write } B;
\end{align*} \quad T_2: \begin{align*}
&\text{read } B; \\
&\text{B := } B - 20; \\
&\text{write } B; \\
&\text{read } C; \\
&\text{C := } C + 20; \\
&\text{write } C;
\end{align*} \]

<table>
<thead>
<tr>
<th>Schedule ( S_1 )</th>
<th>Schedule ( S_2 )</th>
<th>Schedule ( S_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>( T_1 )</td>
<td>( T_1 )</td>
</tr>
<tr>
<td>( T_2 )</td>
<td>( T_2 )</td>
<td>( T_2 )</td>
</tr>
<tr>
<td>read ( A )</td>
<td>read ( A )</td>
<td>read ( A )</td>
</tr>
<tr>
<td>( A - 10 )</td>
<td>( A - 10 )</td>
<td>( A - 10 )</td>
</tr>
<tr>
<td>write ( A )</td>
<td>read ( B )</td>
<td>read ( B )</td>
</tr>
<tr>
<td>read ( B )</td>
<td>( B - 20 )</td>
<td>( B - 20 )</td>
</tr>
<tr>
<td>( B + 10 )</td>
<td>write ( A )</td>
<td>write ( A )</td>
</tr>
<tr>
<td>write ( B )</td>
<td>( B + 10 )</td>
<td>read ( B )</td>
</tr>
<tr>
<td>read ( B )</td>
<td>( C + 20 )</td>
<td>( B + 10 )</td>
</tr>
<tr>
<td>( B - 20 )</td>
<td>write ( B )</td>
<td>read ( C )</td>
</tr>
<tr>
<td>write ( B )</td>
<td>( C + 20 )</td>
<td>write ( B )</td>
</tr>
<tr>
<td>read ( C )</td>
<td>write ( C )</td>
<td>write ( C )</td>
</tr>
<tr>
<td>( C + 20 )</td>
<td>write ( C )</td>
<td>write ( C )</td>
</tr>
</tbody>
</table>
Synchronization and locks

• When is a schedule „fine“?
  – When it is serializable
  – I.e., when it is equivalent to a serial schedule
  – Proof serializability of schedules

• Strategy: Blocking everything is dreadful

• Strategy: Checking after execution is wasteful

• Synchronization protocols
  – Guarantee to produce only serializable schedules
  – Require certain well-behavior of transactions
    • Two phase locking, multi-version synchronization, timestamp synchronization, ...

• Be careful with deadlocks