
Jens Teubner, DBIS Group
jens.teubner@cs.tu-dortmund.de

Winter 2015/16
Part II

Storage: Disks and Files
SQL Commands

Parser
Optimizer
Operator Evaluator
Executor
Files and Access Methods
Buffer Manager
Disk Space Manager
Recovery Manager
Transaction Manager
Lock Manager

DBMS

data files, indices, ...

Database
Memory Hierarchy

- fast, but expensive and small, memory close to CPU
- larger, slower memory at the periphery
- We’ll try to hide latency by using the fast memory as a **cache**.
A stepper motor positions an array of disk heads on the requested track.

Platters (disks) steadily rotate.

Disks are managed in blocks: the system reads/writes data one block at a time.
Access Time

This design has implications on the access time to read/write a given block:

1. Move disk arms to desired track (seek time $t_s$).
2. Wait for desired block to rotate under disk head (rotational delay $t_r$).
3. Read/write data (transfer time $t_{tr}$)

→ access time: $t = t_s + t_r + t_{tr}$
Example: Notebook drive Hitachi Travelstar 7K200

- 4 heads, 2 disks, 512 bytes/sector, 200 GB capacity
- Rotational speed: 7200 rpm
- Average seek time: 10 ms
- Transfer rate: \( \approx 50 \text{ MB/s} \)

What is the access time to read an 8 KB data block?

Average seek time \( t_s = 10 \text{ ms} \)

Average rotational delay:
\[
t_r = \frac{1}{2} \cdot \frac{1}{7200} \approx 4.17 \text{ ms}
\]

Transfer time for 8 KB:
\[
t_{tr} = \frac{8 \text{ KB}}{50 \text{ MB/s}} = 0.16 \text{ ms}
\]

Access time for an 8 KB data block:
\[
t = t_s + t_r + t_{tr} = 14.33 \text{ ms}
\]
Sequential vs. Random Access

**Example:** Read 1000 blocks of size 8 KB

- **random access:**
  \[ t_{\text{rnd}} = 1000 \cdot 14.33 \text{ ms} = 14.33 \text{ s} \]

- **sequential read:**
  \[ t_{\text{seq}} = t_s + t_r + 1000 \cdot t_{tr} + \frac{16 \cdot 1000}{63} \cdot t_{s,\text{track-to-track}} \]
  \[ = 10 \text{ ms} + 4.14 \text{ ms} + 160 \text{ ms} + 254 \text{ ms} \approx 428 \text{ ms} \]

  The Travelstar 7K200 has 63 sectors per track, with a 1 ms track-to-track seek time; one 8 KB block occupies 16 sectors.

→ Sequential I/O is **much** faster than random I/O.

→ **Avoid random I/O** whenever possible.

→ As soon as we need at least \( \frac{428 \text{ ms}}{14330 \text{ ms}} = 3\% \) of a file, we better read the **entire** file!
Performance Tricks

System builders play a number of tricks to improve performance.

**track skewing**
Align sector 0 of each track to avoid rotational delay during sequential scans.

**request scheduling**
If multiple requests have to be served, choose the one that requires the smallest arm movement (SPTF: shortest positioning time first).

**zoning**
Outer tracks are longer than the inner ones. Therefore, divide outer tracks into more sectors than inners.
Hard Disk Latency

Latency [ms]

Date

Evolution of Hard Disk Technology

Disk latencies have only marginally improved over the last years ($\approx 10\%$ per year).

But:

- Throughput (i.e., transfer rates) improve by $\approx 50\%$ per year.
- Hard disk capacity grows by $\approx 50\%$ every year.

Therefore:

- Random access cost hurts even more as time progresses.
Ways to Improve I/O Performance

The latency penalty is hard to avoid.

But:

- Throughput can be increased rather easily by exploiting parallelism.
- Idea: Use multiple disks and access them in parallel.

TPC-C: An industry benchmark for OLTP

Some while ago, the number one system (DB2 9.5 on AIX) used

- 10,992 disk drives (73.4 GB each, 15,000 rpm) (!)
  (plus 8 internal SCSI drives with 146.8 GB each),
- connected with 68 × 4 Gbit Fibre Channel adapters,
- yielding 6 mio transactions per minute.
Disk Mirroring

- Replicate data onto multiple disks

- I/O parallelism only for **reads**.
- Improved failure tolerance (can survive one disk failure).
- This is also known as **RAID 1** (mirroring without parity). (RAID: Redundant Array of Inexpensive Disks)
Disk Striping

- Distribute data over disks

- Full I/O parallelism.
- High failure risk (here: 3 times risk of single disk failure)!
- Also known as **RAID 0** (striping without parity).
Disk Striping with Parity

- Distribute data and parity information over disks.

- High I/O parallelism.
- Fault tolerance: one disk can fail without data loss (two disks with dual parity/RAID 6).
- Also known as **RAID 5** (striping with distributed parity).
Solid-State Drives: Technology

**Basis:** MOS transistor

---

**Flash cell:**

- Add (fully isolated) **floating gate** in-between.
- Charge on floating gate shifts characteristics of the source/control gate/drain transistor.
  - Use to “read” charge state
- (Dis-)charging of floating gate only through high voltage (tunnel effect)
  - Charge “trapped” → persistence

Solid-State Drives: Technology

Miniaturization:
- Combine many cells to achieve tight packing
  - NAND Flash
  - Must read blocks of data at once (≈ hard disks)
- Single-level cells (SLC) vs. Multi-level cells (MLC)
  - Cost/density ↔ reliability trade-off

Challenges:
- Feature size ↘ ⇒ reliability ↘
  - Fewer electrons of charge, thinner isolation layers
  - Limited retention
- Over time, writes damage isolation layer
  - Limited endurance ($10^4 \sim 10^5$ writes per cell)
- Block based erasure (→ no update in place)
  - Write amplification, slow writes
Solid-state drives (SSDs) as an alternative to conventional hard disks?

- SSDs provide **very low-latency random read access**.
- **Random writes**, however, are significantly **slower** than on traditional magnetic drives.
  - Pages have to be **erased** before they can be updated.
  - Once pages have been erased, sequentially writing them is almost as fast as reading.
- Adapting databases to these characteristics is a current research topic.

Samsung 32 GB flash disk; 4096 bytes read/written randomly. Source: Koltsidas and Viglas. Flashing up the Storage Layer. VLDB 2008.
Phase-Change Memory: Physics

**History of Phase-change memory**
- Late 1960s – Ovsishchyns shows reversible electrical switching in disordered semiconductors
- Early 1970s – Much research on mechanisms, but everything was too slow!

**Electrical conductivity**
- **Crystalline phase**
  - Low resistance
  - High reflectivity
- **Amorphous phase**
  - High resistance
  - Low reflectivity

**Phase-Change RAM**
- Access device (transistor, diode)
- PCRAM: programmable resistor

**Potential headache:**
- High power/current affects scaling!
- If crystallization is slow affects performance!

Phase-Change Memory: Technology

History of Phase-change memory

- late 1960’s – Ovsishchys shows reversible electrical switching in disordered semiconductors
- early 1970’s – much research on mechanisms, but everything was too slow!

Electrical conductivity

<table>
<thead>
<tr>
<th>Crystalline phase</th>
<th>Amorphous phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low resistance</td>
<td>High resistance</td>
</tr>
<tr>
<td>High reflectivity</td>
<td>Low reflectivity</td>
</tr>
</tbody>
</table>

Phase-Change RAM

- Access device (transistor, diode)
- PCRAM “programmable resistor”

Potential headache:
- High power/current → affects scaling!
- If crystallization is slow → affects performance!


© Jens Teubner · Architecture & Implementation of DBMS · Winter 2015/16

28
Storage-Class Memory (SCM)

Phase-Change Memory is one promising technology to realize *storage-class memory*:

- **Persistent** (like disks or SSDs)
- **RAM-like access characteristics**
  → Speed-wise, but also with byte-level addressing

First prototypes exist!

**Challenges/Questions:**

- How **scalable** are SCM technologies? (so far looks good)
- How can SCM be **integrated** into a system?
  → Access SCM like a block device or like RAM?
- What does fast, byte-addressable storage mean for software?
  → *E.g.*, database recovery mechanisms
Network-Based Storage

The network is **not** a bottleneck any more:

- Hard disk: 50–100 MB/s
- Serial ATA: 375 MB/s (600 MB/s soon)
  Ultra-640 SCSI: 640 MB/s
- 10 gigabit Ethernet: 1,250 MB/s (latency: \(\sim \mu s\))
  Infiniband QDR: 12,000 MB/s (latency: \(\sim \mu s\))

for comparison:
- PC2-5300 DDR2-SDRAM (dual channel): 10.6 GB/s
- PC3-12800 DDR3-SDRAM (dual channel): 25.6 GB/s

→ Why not use the network for database storage?
Storage Area Network

- **Block-based** network access to storage
  - Seen as logical disks (“give me block 4711 from disk 42”)
  - Unlike network file systems (e.g., NFS, CIFS)
- SAN storage devices typically abstract from RAID or physical disks and present logical drives to the DBMS
  - Hardware acceleration and simplified maintainability
- Typically local networks with multiple servers and storage resources participating
  - Failure tolerance and increased flexibility
Grid or Cloud Storage

Some big enterprises employ clusters with **thousands** of commodity PCs (e.g., Google, Amazon):

- **system cost** $\leftrightarrow$ **reliability** and **performance**,
- use **massive replication** for data storage.

Spare CPU cycles and disk space can be sold as a **service**.

**Amazon’s “Elastic Computing Cloud (EC2)”**

Use Amazon’s compute cluster by the hour ($\sim 10 \, \text{¢/hour}$).

**Amazon’s “Simple Storage Systems (S3)”**

“Infinite” store for objects between 1 Byte and 5 GB in size, with a simple key $\leftrightarrow$ value interface.

- Latency: 100 ms to 1 s (not impacted by load)
- pricing $\approx$ disk drives (but addl. cost for access)

→ **Build a database on S3?** (↗ Brantner et al., SIGMOD 2008)
Managing Space

Web Forms → SQL Commands → Executor → Files and Access Methods

Applications → SQL Commands → Parser → Operator Evaluator

SQL Interface → SQL Commands → Optimizer → Lock Manager

DBMS

Transaction Manager

Lock Manager

Buffer Manager

Disk Space Manager

data files, indices, ...

Recovery Manager

Database

Jens Teubner · Architecture & Implementation of DBMS · Winter 2015/16
Managing Space

The **disk space manager**

- abstracts from the gory details of the underlying storage
- provides the concept of a **page** (typically 4–64 KB) as a unit of storage to the remaining system components
- maintains the mapping

\[
\text{page number} \mapsto \text{physical location}
\]

where a physical location could be, *e.g.*,

- an OS file name and an offset within that file,
- head, sector, and track of a hard drive, or
- tape number and offset for data stored in a tape library
The disk space manager also keeps track of used/free blocks.

1. Maintain a linked list of free pages
   - When a page is no longer needed, add it to the list.

2. Maintain a bitmap with one bit for each page
   - Toggle bit \( n \) when page \( n \) is (de-)allocated.

To exploit **sequential access**, it may be useful to allocate **contiguous** sequences of pages. **Which of the techniques 1 or 2 would you choose to support this?**
The **buffer manager**

- mediates between external storage and main memory,
- manages a designated main memory area, the **buffer pool** for this task.

Disk pages are brought into memory as needed and loaded into memory **frames**.

A **replacement policy** decides which page to evict when the buffer is full.
Interface to the Buffer Manager

Higher-level code requests (pins) pages from the buffer manager and releases (unpins) pages after use.

**pin** *(pageno)*
Request page number *pageno* from the buffer manager, load it into memory if necessary. Returns a reference to the frame containing *pageno*.

**unpin** *(pageno, dirty)*
Release page number *pageno*, making it a candidate for eviction. Must set *dirty = true* if page was modified.

Why do we need the *dirty* bit?
Implementation of pin()

1. Function: \( \text{pin}(pageno) \)
2. if buffer pool already contains \( pageno \) then
   3. \( \text{pinCount}(pageno) \leftarrow \text{pinCount}(pageno) + 1 \); 
   4. return address of frame holding \( pageno \); 
else
   5. select a victim frame \( v \) using the replacement policy;
      6. if \( \text{dirty}(v) \) then
         7. write \( v \) to disk;
         8. read page \( pageno \) from disk into frame \( v \);
         9. \( \text{pinCount}(pageno) \leftarrow 1 \);
         10. \( \text{dirty}(pageno) \leftarrow \text{false} \);
      11. return address of frame \( v \);
Implementation of \texttt{unpin()} 

1. \textbf{Function: } \texttt{unpin(pageno, dirty)}
2. \texttt{pinCount(pageno) \leftarrow \text{pinCount}(pageno) - 1;}
3. \texttt{if dirty then}
4. \hspace{1em} \texttt{dirty(pageno) \leftarrow dirty;}

\begin{itemize}
  \item Why don’t we write pages back to disk during \texttt{unpin()}?
\end{itemize}
The effectiveness of the buffer manager’s **caching** functionality can depend on the **replacement policy** it uses, *e.g.*, 

**Least Recently Used (LRU)**

Evict the page whose latest `unpin()` is longest ago.

**LRU-\(k\)**

Like LRU, but considers \(k\)-latest `unpin()`, not just latest.

**Most Recently Used (MRU)**

Evict the page that has been unpinned most recently.

**Random**

Pick a victim randomly.

What could be the rationales behind each of these strategies?
Buffer Management in Reality

Prefetching

Buffer managers try to anticipate page requests to overlap CPU and I/O operations.

- **Speculative prefetching**: Assume sequential scan and automatically read ahead.
- **Prefetch lists**: Some database algorithms can instruct the buffer manager with a list of pages to prefetch.

Page fixing/hating

Higher-level code may request to **fix** a page if it may be useful in the near future (e.g., index pages).
Likewise, an operator that **hates** a page won’t access it any time soon (e.g., table pages in a sequential scan).

Partitioned buffer pools

*E.g.*, separate pools for indexes and tables.
Hmm... Didn’t we just re-invent the operating system?

Yes,

- disk space management and buffer management very much look like file management and virtual memory in OSs.

But,

- a DBMS may be much more aware of the access patterns of certain operators (→ prefetching, page fixing/hating),
- transaction management often calls for a defined order of write operations,
- technical reasons may make OS tools unsuitable for a database (e.g., file size limitation, platform independence).
In fact, databases and operating systems sometimes interfere.

- Operating system and buffer manager effectively buffer the same data twice.
- Things get really bad if parts of the DBMS buffer get swapped out to disk by OS VM manager.
- Therefore, databases try to turn off OS functionality as much as possible.
  → Raw disk access instead of OS files.

(Similar story: DBMS TX management vs. journaling file systems.)
Files and Records

SQL Commands

Parser  Optimizer
Executor  Operator Evaluator

Files and Access Methods

Transaction Manager
Lock Manager
Buffer Manager
Disk Space Manager

Recovery Manager

DBMS

Database

data files, indices, …
So far we have talked about **pages**. Their management is oblivious with respect to their actual content.

On the conceptual level, a DBMS manages **tables of tuples** and **indexes** (among others).

Such tables are implemented as **files of records**:

- A **file** consists of **one or more pages**.
- Each **page** contains **one or more records**.
- Each **record** corresponds to **one tuple**.
The most important type of files in a database is the heap file. It stores records in no particular order (in line with, e.g., SQL).

Linked list of pages

+ easy to implement
  - most pages will end up in free page list
  - might have to search many pages to place a (large) record
Heap Files

Directory of pages

- use as **space map** with information about free page
  - granularity as trade-off space ↔ accuracy
    (range from *open/closed* bit to exact information)

+ free space search more efficient
- small memory overhead to host directory
Which page to pick for the insertion of a new record?

**Append Only**
Always insert into last page. Otherwise, create a new page.

**Best Fit**
Reduces fragmentation, but requires searching the entire space map for each insert.

**First Fit**
Search from beginning, take first page with enough space.
(→ These pages quickly fill up, and we waste a lot of search effort in first pages afterwards.)

**Next Fit**
Maintain `cursor` and continue searching where search stopped last time.
Free Space Witnesses

We can accelerate the search by remembering **witnesses**:

- Classify pages into **buckets**, e.g., “75 %–100 % full”, “50 %–75 % full”, “25 %–50 % full”, and “0 %–25 % full”.
- For each bucket, remember some **witness pages**.
- Do a regular best/first/next fit search only if no witness is recorded for the specific bucket.
- Populate witness information, e.g., as a side effect when searching for a best/first/next fit page.
**Inside a Page**

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>SEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>4711</td>
<td>John</td>
<td>M</td>
</tr>
<tr>
<td>1723</td>
<td>Marc</td>
<td>M</td>
</tr>
<tr>
<td>6381</td>
<td>Betty</td>
<td>F</td>
</tr>
</tbody>
</table>

- **record identifier (rid):**  
  \( \langle \text{pageno}, \text{slotno} \rangle \)

- **record position (within page):**  
  \( \text{slotno} \times \text{bytes per slot} \)

- **Tuple deletion?**
  - record id shouldn’t change  
  → **slot directory** (bitmap)

![Diagram showing record identifiers and position within a page](image)
Inside a Page—Variable-Sized Fields

- Variable-sized fields moved to **end** of each record.
  - Placeholder points to location.
  - **Why?**

- Slot directory points to start of each record.

- Records **can move** on page.
  - *E.g.*, if field size changes.

- Create “**forward address**” if record won’t fit on page.
  - **Future updates?**
In DB2, the slot directory grows from the front, data grows from the end:

BPS Page Header:
- Page Data Offset = 48
- Page Data Length = 4048
- Page LSN = 0000 0438 8F85

Object Type = Data Object

Data Page Header:
- Slot Count = 103
- Total Free Space = 48
- Free Space Offset = 216
- Maximum Record Size = 37

Data Records:

Slot 0:
- Offset Location = 3991 (xF97)
- Record Length = 37 (x25)
- Record Type = Table Data Record (FIXEDVAR) (PUNC)

... Slot 1:
- Offset Location = 3954 (xF72)
- Record Length = 37 (x25)
- Record Type = Table Data Record (FIXEDVAR) (PUNC)

- Such data can be obtained with db2dart.
- Observe how slot 4 is marked ‘deleted’ (FFFF).
Sparse Columns

An alternative is **interpreted storage**.

<table>
<thead>
<tr>
<th>Interpreted Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨ID, 4711⟩, ⟨NAME, John⟩, ⟨SEX, M⟩</td>
</tr>
<tr>
<td>⟨ID, 1723⟩, ⟨NAME, Marc⟩, ⟨SEX, M⟩</td>
</tr>
<tr>
<td>⟨ID, 6381⟩, ⟨NAME, Betty⟩, ⟨SEX, F⟩</td>
</tr>
</tbody>
</table>

⚠️ Why would one want to do this?
Microsoft SQL Server 2008 provides support for sparse columns.

Columns marked as SPARSE are put into an interpreted storage.

```sql
CREATE TABLE Products
(···, Card VARCHAR(10) SPARSE NULL, ···)
```

The internal storage is designed for fast access.

<table>
<thead>
<tr>
<th>Interpreted Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Header</strong></td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>4 ··· 100</td>
</tr>
<tr>
<td>101 ··· 200</td>
</tr>
<tr>
<td>201 ··· 300</td>
</tr>
</tbody>
</table>

Alternative Page Layouts

We have just populated data pages in a **row-wise** fashion:

We could as well do that **column-wise**:
These two approaches are also known as **NSM** (*n*-ary storage model) and **DSM** (decomposition storage model).\(^1\)

- Tuning knob for certain workload types (e.g., OLAP)
- Different behavior with respect to **compression**.

A hybrid approach is the **PAX** (Partition Attributes Accross) layout:

- Divide each page into **minipages**.
- Group attributes into them.


\(^1\)Recently, the terms **row-store** and **column-store** have become popular, too.
Recap

Magnetic Disks

Random access orders of magnitude slower than sequential.

Disk Space Manager

Abstracts from hardware details and maps page number $\mapsto$ physical location.

Buffer Manager

Page caching in main memory; \texttt{pin ()/unpin ()} interface; \texttt{replacement policy} crucial for effectiveness.

File Organization

Stable record identifiers (rids); maintenance with fixed-sized records and variable-sized fields; NSM vs. DSM.