Databases

Storage & Indexing

Pierre Senellart

PSL UNIVERSITÉ PARIS

25 March 2026

Outline

Why study storage in a database course?

- A DBMS must manage large amounts of data
- This data must be stored on persistent storage
- Query performance is often dominated by data access, not by arithmetic, comparisons, and other computation
- Therefore, the physical organization of data is a central part of database systems

How to compare memory areas

To compare different storage technologies, the main criteria are:

Capacity: how much data can be stored

Latency: the delay before the first byte of data becomes available

Bandwidth: the rate at which data can be transferred once the transfer has started

Page size: the typical granularity in which data is transferred between this memory area and the rest of the system

- A memory area may have low latency but small capacity
- Another may have large capacity but poor latency or bandwidth
- Page size matters because DBMSs usually move data in blocks, not byte by byte

Primary vs secondary storage

- Memory hierarchy in computers:

  CPU registers: extremely small (hundreds of bytes), extremely fast (< 1 ns latency), volatile

  CPU cache: very small (KB to MB), very fast (1–10 ns latency), volatile

  Main memory (RAM): limited capacity (GB), fast (10–100 ns latency), volatile

  Secondary storage (SSD, disk): much larger (TB), much slower (50 µs to 15 ms latency), persistent

- A DBMS relies on secondary storage because:
  - databases are often larger than RAM
  - data must survive program termination and machine reboot

A running idea

- For large databases:
  - CPU time matters
  - algorithmic complexity matters
  - but I/O cost often dominates
- So we need a model answering questions such as:
  - How expensive is a random access (accessing an arbitrary tuple in the database)?
  - How expensive is a sequential scan (accessing all tuples stored together)?
  - How many blocks of data must be read?

Outline

Magnetic hard disk drives (HDDs)

- A magnetic hard drive stores data on rotating platters by changing the magnetic orientation of tiny areas of the disk
- To read some data, the drive must first move the head to the right track (seek time)
- Then it must wait until the desired sector rotates under the head (rotational latency)
- Finally, it transfers the data (transfer time)

On HDDs, random access is expensive

Sequential vs random access on HDDs

- If many tuples are stored close to each other:
  - they can often be read sequentially
  - one seek and one rotational delay are amortized over a large transfer
- If they are scattered:
  - the system may incur many random accesses
  - each access pays seek time and rotational latency again
- Locality of data matters
- The disk controller retrieves data by whole blocks/pages, e.g., a few kilobytes at a time

Solid-state drives (SSDs)

- SSDs have no moving mechanical parts, they use Flash memory (itself based on floating-gate transistors, which are able to keep an electronic charge unchanged for a long period of time)
- Compared with HDDs:
  - they have much lower latency, especially for random reads
  - they are still much slower than RAM
  - they are still accessed in blocks/pages, not tuple by tuple

SSDs reduce the penalty of random access, but locality still matters

Orders of magnitude of memory types

Type	Cap. (typical)	Latency	Bandwidth	Page size	Price / TB (€)
RAM	32 GB	10–100 ns	10–500 GB/s	64 B	4,000–18,000 €
SSD	1 TB	50–1000 µs	0.5–3.5 GB/s	4 KiB	100–200 €
HDD	4 TB	5–15 ms	100–300 MB/s	4 KiB	35–40 €

- These are only orders of magnitude and typical values: exact values depend on the machine, the device, and the market segment
- The price / TB figures are especially rough, but they help explain why RAM is precious, SSDs are affordable but not cheap, and HDDs remain attractive for bulk storage

Outline

A simple storage cost model

A first approximation is:

$$\text{time} \approx \text{latency} + \frac{\text{data size}}{\text{bandwidth}}$$

- latency: fixed initial cost before data starts arriving
- bandwidth or throughput: transfer rate once data starts arriving

Consequences:

- one large read is often better than many tiny reads
- sequential access is often much better than random access

Cost model for HDDs

For magnetic disks, one often decomposes:

$$\text{access time} \approx \text{seek time} + \text{rotational latency} + \text{transfer time}$$

- the first two terms dominate for small random accesses
- the last term dominates for large sequential transfers

This is one of the reasons why DBMSs store data in large blocks

Why DBMSs reason in pages

A DBMS usually does not reason in terms of single tuples on disk. Instead:

- data is transferred in fixed-size pages (or blocks)
- A page/block is typically at least the size of a disk block, possibly larger (e.g., 16 KB by default in MySQL)
- a relation is stored as a file made of many pages
- the cost of a plan is often estimated in terms of page reads and writes

The basic cost unit is often one page I/O

Tuple-by-tuple vs page-by-page

- Suppose a page contains 100 tuples.
  - Reading 100 tuples one by one would incur many accesses
  - Reading one page gets all 100 tuples at once
  - This is true even if the query will later inspect tuples individually
- Storage devices and DBMSs naturally work at the page level, not at the tuple level

Outline

Files, pages, and records

Common (but not the only possible) storage mechanism:

- A relation is stored in a file
- A file is divided into pages
- A page contains records (or tuples)

Typical page structure

A page usually contains:

- a page header
- a set of records
- some free space

The header typically stores metadata such as:

- the number of records
- the amount of free space
- metadata about each record and pointers to individual records

Fixed-length and variable-length records

Two common cases:

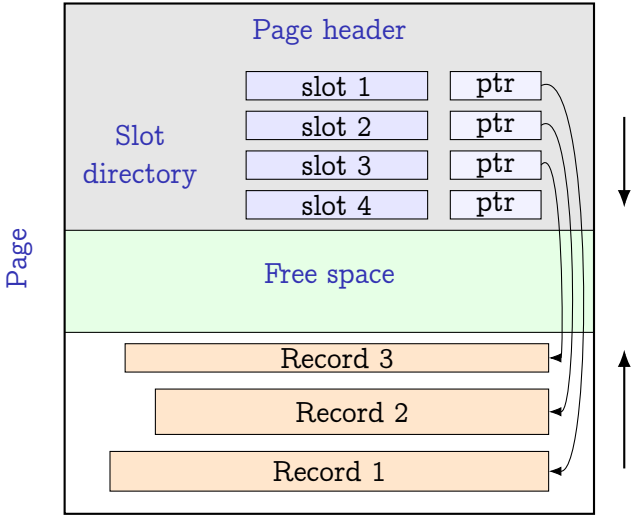
Fixed-length records: easy to locate and simple to manage

Variable-length records: more flexible, but need offsets or indirection

Variable-length records arise naturally with:

- strings of unbounded size
- optional attributes
- more complex values

Slotted pages



- A common solution is the slotted-page organization.
- A slotted page contains:
  - a header
  - a slot directory
  - the records themselves
- Advantages:
  - records can move inside the page
  - external references can still use the same slot number

Record identifiers

- A stored record can often be identified by:

$$(\text{page identifier}, \text{slot identifier})$$

- This is often called a RID or TID
- Used internally by the DBMS to uniquely identify a given tuple (not meant to be used by the user of the DBMS)
- Indexes use TIDs to point to records compactly
- Records can be located quickly once the correct page is known

Heap files

- A heap file stores records (grouped within pages of fixed size) with no particular order
- Advantages:
  - simple
  - inserts are easy
- Disadvantages:
  - searching for a value may require scanning many pages

Sorted files

- A sorted file stores records ordered by some attribute (usually, the primary key)
- Advantages:
  - good for range queries (involving inequalities)
  - searching is easier than in a heap file
- Disadvantages:
  - inserts and deletes are more expensive
  - maintaining order has a cost

Search vs insert

Organization	Search	Insert
Heap file	expensive	cheap
Sorted file	cheaper	more expensive

This kind of trade-off is what motivates indexes

The buffer pool

- A DBMS usually keeps some pages in RAM in a buffer pool
- Limited capacity (128 MB by default in MySQL)
- Pages read most frequently or most often accessed are kept in the buffer pool by the DBMS
- Reading a page already in memory is cheap (cache hit)
- Reading a page not in memory may require disk I/O (cache miss)
- A modified in-memory page may have to be written back later (dirty page)
- Storage cost depends not only on the file layout, but also on what is already in memory

Outline

Why indexes?

Suppose we want to find:

all tuples with `id = 3`

in a large heap file.

- Without an index, the DBMS may need to scan many or all pages
- With an index, an auxiliary structure can guide the search much more quickly

What is an index?

- An index is an auxiliary data structure that helps locate tuples.
- Typically, it stores:
  - a search key value
  - a reference to the corresponding record or page
- An index is useful because:
  - it is much smaller than the whole table (may be kept in the buffer pool!)
  - it is organized for efficient lookup

Indexes are not free

- Benefits:
  - faster search
  - better support for common queries
- Costs:
  - extra storage space
  - inserts, deletes, and updates must also maintain the index

Indexes speed up reads, but make updates more expensive

Dense vs sparse indexes

Dense index: one entry for every record or every key occurrence

Sparse index: fewer entries, for instance one entry per page range

Sparse indexes are usually useful when the underlying data is already stored in sorted order.

Clustered vs unclustered indexes

Clustered index: the table itself is stored in the same order as the index key

Unclustered index: the table is stored in a different order

This matters a lot for range queries:

- clustered indexes preserve locality (once we found the beginning of a range, we can keep navigating the tuples of the table in the sorted file)
- unclustered indexes may lead to many scattered page reads

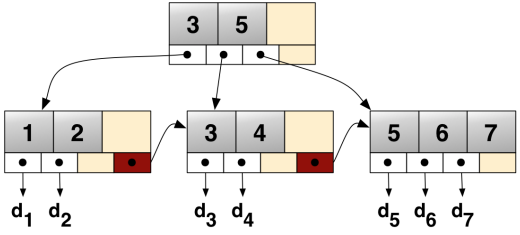
Outline

Why not binary search trees?

- In RAM, binary search trees can be a good indexing structure
- On disk, they are a bad idea:
  - one node per page would give a tall tree
  - many levels mean many page accesses
- On disk, we prefer:
  - large nodes
  - high fan-out
  - small height

B^+-trees

A B^+-tree is a balanced search tree designed for block storage.



- Similar in principle to a binary search tree (except non-binary)
- One node usually fits in one page (high fan-out)
- Internal nodes contain separator keys and child pointers
- Leaves contain keys and pointers to records or pages
- All leaves are at the same depth

Why B$^+$-trees work well on disk

Because nodes are large:

- each internal node can have many children
- therefore the branching factor is high
- therefore the height is small

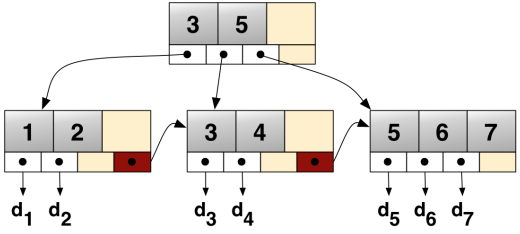
So even for large relations, a lookup often requires only a few page reads.

A rough order-of-magnitude calculation

- Suppose:
  - one node fits in one page
  - each internal node has about 100 children
- Then:
  - height 1: about 10^2 leaves
  - height 2: about 10^4 leaves
  - height 3: about 10^6 leaves
  - height 4: about 10^8 leaves
- More generally:

$$\text{depth} = \Theta(\log_{\text{fan-out}}(\#\text{tuples})) \quad \text{or} \quad \#\text{tuples} = \Theta(\text{fan-out}^{\text{depth}})$$

Search in a B^+-tree



- To search for a key:
  - start at the root
  - choose the correct child according to separator keys
  - continue until a leaf is reached
  - in the leaf, find the corresponding record or page pointer
- Typical cost:
  - about one page read per tree level
  - plus maybe one data page read

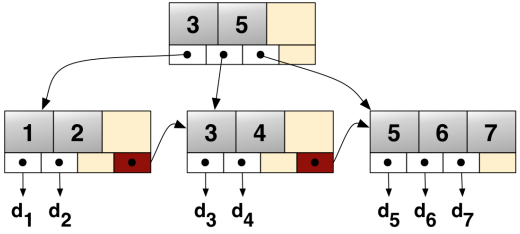
Insertions in a B^+-tree

To insert a new key (sketch):

- find the correct leaf
- insert the key if the leaf has room
- otherwise, split the leaf
- if necessary, propagate the split upward

This is why B^+-trees remain balanced.

Why B$^+$-trees are good for range queries



Leaves of a B$^+$-tree are ordered by key, and usually linked from one leaf to the next
So to answer a range query:

- find the first relevant leaf
- scan consecutive leaves until the range ends

B$^+$-trees support both equality queries and range queries

Outline

Hash indexes

- Another family of indexes is based on hashing
- Main idea:
  - apply a hash function to the search key
  - use the result to locate a bucket
  - the bucket contains pointers to matching tuples
- In contrast with hash tables used for in-memory indexing, optimized for bucket sizes of around one page

Why hash indexes are useful

Hashing is very good for:

exact-match queries

For instance:

```
id = 3
```

because the hash function sends the search directly to the relevant bucket.

Buckets and overflow

- If several keys map to the same bucket, we have a collision.
- Consequences may include:
  - several entries in the same bucket
  - overflow pages if the bucket becomes too full
- So hash files must manage:
  - collisions
  - load factor
  - growth of the file

Why hash indexes are bad for range queries

Hashing destroys the order of keys.

So a query such as

$$10 \leqslant \texttt{id} \leqslant 20$$

does not correspond to a contiguous region of the index.

Hash indexes are good for equality queries, but not for range queries

Outline

Equality search: no index vs B^+-tree vs hash

Suppose we search for:

```
id = 3
```

- heap file, no index: possibly scan many or all pages
- B^+-tree index: read a few index pages, then one data page
- hash index: go directly to one bucket, then maybe one data page

Hashing is often the best choice for exact-match queries.

Range search: no index vs B$^+$-tree vs hash

Suppose we search for:

$$10 \leqslant \texttt{id} \leqslant 20$$

- heap file, no index: possibly scan many or all pages
- B$^+$-tree index: locate the first value, then scan consecutive leaves
- hash index: not adapted, because nearby keys are not nearby in the index

B$^+$-trees are much better for range queries.

Role of clustering

Suppose an index returns many matching tuples.

- If the index is clustered, the corresponding tuples are physically close
- If the index is unclustered, the tuples may be spread over many pages

So even with the same logical index, the physical storage of the table still matters.

Summary table

Structure	Equality	Range
Heap file	poor	poor
Sorted file	good	good
B^+-tree	good	very good
Hash index	very good	poor

- Note that sorted files only work for search on the attribute according to which the file is sorted (usually the primary key)
- B^+-tree can be constructed for the primary key or part of it (clustered index) or for other attributes
- There is no universally best organization: it depends on the workload, i.e., on the kind of query we will encounter

What the DBMS optimizer needs

- To choose a good plan, the DBMS must know:
  - what files and indexes exist
  - how data is physically organized
  - approximate costs of scans and index lookups
- Storage and indexing are central to query optimization

Outline

Indexes in MySQL: overview

- In MySQL, indexes are used to find rows with specific values quickly
- How MySQL stores data and available indexes depend on the storage engine used by MySQL (we discuss the default one, InnoDB, but see `SHOW ENGINES`)
- The following indexes are used:
  - Ordinary indexes, `PRIMARY KEY`, and `UNIQUE` indexes are typically based on B^+-trees
  - `FULLTEXT` indexes are specific indexes used for full-text search
  - `SPATIAL` indexes are specific indexes used for geometric data
  - Hash indexes are not generally used in MySQL, but they are used in other DBMSs

Clustered and secondary indexes in MySQL

- In MySQL, the data of a table is organized by the clustered index
- The clustered index is the index on the primary key
- The leaf pages of that index contain the full rows
- Other indexes are called secondary indexes
- A secondary-index entry contains:
  - the indexed columns
  - the primary key of the row
- So a secondary-index lookup often means:
  1. find the primary key in the secondary index
  2. use that primary key to find the row in the clustered index

A consequence: short primary keys are good

Because MySQL stores the primary key in every secondary index entry:

- a long primary key makes all secondary indexes larger
- larger secondary indexes mean:
  - more storage space
  - fewer entries per page
  - potentially more I/O

This is one reason why surrogate keys such as an integer `AUTO_INCREMENT` primary key are often convenient when no adequate primary key exists

Creating indexes in MySQL

Typical syntax:

SQL

```sql
CREATE TABLE Student(
  id INT PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(256) UNIQUE,
  program VARCHAR(50)
);

CREATE INDEX idx_name ON Student(name);
ALTER TABLE Student ADD INDEX idx_program(program);
```

Common kinds of indexes:

- `PRIMARY KEY`
- `UNIQUE`
- ordinary `INDEX`

Seeing index usage with EXPLAIN

MySQL can show how it plans to execute a query.

SQL

```sql
EXPLAIN
SELECT *
FROM Student
WHERE id = 3;
```

Important columns in traditional EXPLAIN output:

- possible_keys: indexes that could be used
- key: the index actually chosen

A small example

Without an index on program, the query

SQL

```
SELECT *
FROM Student
WHERE program = 'CS';
```

may require a large scan.

After creating an index:

SQL

```
CREATE INDEX idx_program ON Student(program);
```

MySQL may use that index instead.

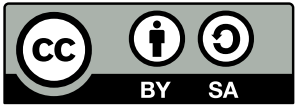
Indexes are useful only if the optimizer decides that using them is cheaper than scanning the table.

Outline

Conclusion

- Storage is central because databases are persistent and often larger than RAM
- The main physical cost is often page I/O
- DBMSs store relations in files made of pages
- File organization already affects performance
- Indexes trade extra space and update cost for faster search
- B^+-trees are versatile and support range queries well
- Hash indexes are excellent for exact-match queries
- Indexes are one of the tools used by the query optimizer (see later lecture on this)

Licensing



Media used:

- Hard drive picture slide 8 is CC-BY-SA 4.0 by Bubba73
- B^+-tree picture slides 35, 38, and 40 is CC-BY 3.0 by Grundprinzip