Database Internals

23-09-07

Org stuff

Only labs matter for the final score. There will be quizzes.

When you work on a task `k` in project `n`, you need to:

- Create a branch named `task<k>-<mnemonic-suffix>-<last-name>`, e.g. `task0-warmup-ivanov`
- Write code + `git commit` + `git push`
- Create a PR in project `n` repo and assign a teammate as a reviewer

Recap

A **relational database** organizes data into structured tables (two-dimensional structures) with rows and columns, allowing for efficient storage, retrieval, and manipulation of data.

Indexes are data structures used in tables, improving the speed of data retrieval operations. They allow to add, remove, sort or get the rows without scanning the whole table. Common types used for implementing indexes are B-tree (balanced tree) and hash table.

Basic concepts of transactions:

- *Atomicity*. Transaction is treated as a single, indivisible unit of work If a transaction fails at any point, it is rolled back to its previous state.
- *Consistency*. This means that the integrity constraints of the database (e.g., primary keys, foreign keys) should not be violated during a transaction.
- *Isolation*. Changes made by one transaction should not be visible to other transactions until the first transaction is committed. This ensures that concurrent transactions do not interfere with each other.
- *Durability*. Once a transaction is committed, its changes should be permanent and survive system failures. The database should be able to recover to a consistent state after a crash.

SQL commands

SQL is a domain-specific language used to interact with relational databases.

Example of creating tables:

```sql
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(50)
);

CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    total_amount DECIMAL(10, 2)
);

-- Inserting some sample data into the "customers" table
INSERT INTO customers (customer_id, customer_name)
VALUES
    (1, 'Alice'),
    (2, 'Bob'),
    (3, 'Charlie'),
    (4, 'David');

-- Inserting some sample data into the "orders" table
INSERT INTO orders (order_id, customer_id, order_date, total_amount)
VALUES
    (101, 1, '2023-01-05', 250.00),
    (102, 2, '2023-01-10', 120.50),
    (103, 1, '2023-01-15', 300.75),
    (104, 3, '2023-01-20', 450.20);
```

Customers

customer_id	customer_name
1	Alice
2	Bob
3	Charlie
4	David

Orders

order_id	customer_id	order_date	total_amount
101	1	2023-01-05	250
102	2	2023-01-10	120.5
103	1	2023-01-15	300.75
104	3	2023-01-20	450.2

`JOIN` command is used to combine rows from two or more tables based on a related column between them.

- `INNER JOIN` returns the rows that have matching values in both tables:

```sql
SELECT orders.order_id, customers.customer_name
FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id;
```

```
| order_id | customer_name |
|----------|---------------|
| 101      | Alice         |
| 102      | Bob           |
| 103      | Alice         |
| 104      | Charlie       |
```

- `LEFT JOIN` returns all rows from the left table and the matched rows from the right table. If there is no match in the right table, `NULL` value is used:

```sql
SELECT customers.customer_name, orders.order_id
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id;
```

```
| customer_name | order_id |
|---------------|----------|
| Alice         | 101      |
| Bob           | 102      |
| Alice         | 103      |
| Charlie       | 104      |
| David         | NULL     |
```

- `RIGHT JOIN` (or `RIGHT OUTER JOIN`) returns all rows from the right table and the matched rows from the left table. If there is no match in the left table, NULL values are returned for columns from the left table:

```sql
SELECT customers.customer_name, orders.order_id
FROM customers
RIGHT JOIN orders ON customers.customer_id = orders.customer_id;
```

```
| customer_name | order_id |
|---------------|----------|
| Alice         | 101      |
| Bob           | 102      |
| Alice         | 103      |
| Charlie       | 104      |
```

- `FULL OUTER JOIN` returns all rows when there is a match in either the left or right table. If there is no match, NULL values are returned for columns from the table with no match:

```sql
SELECT customers.customer_name, orders.order_id
FROM customers
FULL OUTER JOIN orders ON customers.customer_id = orders.customer_id;
```

```
| customer_name | order_id |
|---------------|----------|
| Alice         | 101      |
| Bob           | 102      |
| Alice         | 103      |
| Charlie       | 104      |
| David         | NULL     |
```

The `WHERE` clause is used to filter rows based on specified conditions:

```sql
SELECT orders.order_id, customers.customer_name
FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id
WHERE orders.order_date >= '2023-01-12';
```

```
| order_id | customer_name |
|----------|---------------|
| 103      | Alice         |
| 104      | Charlie       |
```

Secondary storage

Primary storage is what CPU can reach ”directly”, e.g. cache or RAM. Secondary storage is accessed with controllers and it is where all our data resides permanently.

There are several data storage characteristics:

- how much data can be stored (capacity)
- how fast a random access is (random access latency)
- how much data per second can be read or written (transfer rate)

- what happens if electricity switches off (volatility)
- how much does it cost to store 1Gb (price)

	Capacity	Latency	T/Rate	Price
L1-L3	32 kb - 96 Mb	1-10 ns	400-3000 Gb/s	$ ..50k/Gb
RAM	2 Gb-40 Tb	10-20 ns	10-20 Gb/s	$ 5/Gb
HDD	..30 Tb	6-20 ms	100-300 Mb/s	$ 0.015/Gb
SSD	..30 Tb	0.07..0.25 ms	0.5-5 Gb/s	$ 0.03-0.1/Gb
Tape	..20 Tb	:)	400 Mb/s	$ 0.005/Gb

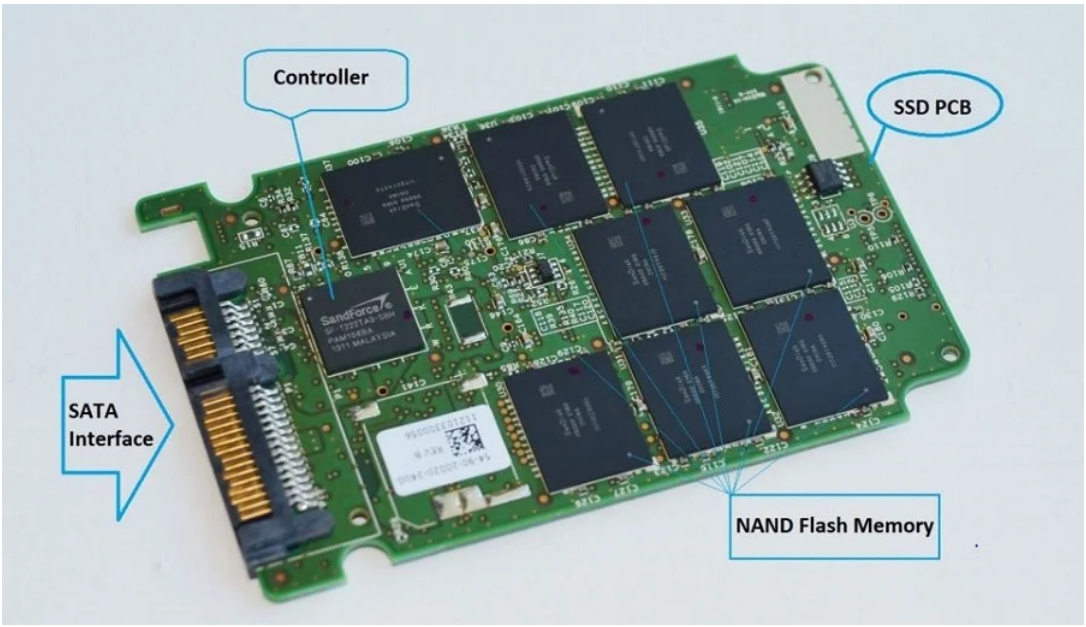
HDD

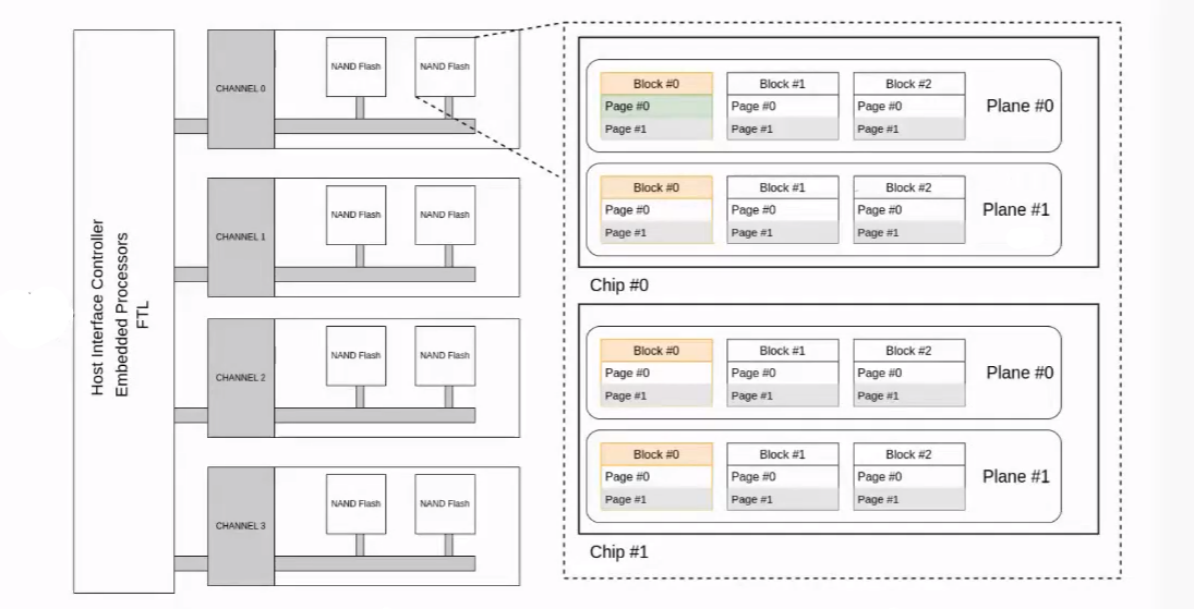
A disk consists of several *platters*. A *track* is a circle on a platter. Tracks located one above other on different platters form a *cylinder*. A *sector* is a fragment of the track.

In the HDD, data is stored on a **rotating magnetic disk** which is divided into tracks, sectors and cylinders. An electromagnet in the read/write head charges the disk's surface with either a positive or negative charge, this is how binary 1 or 0 is represented.

The **read/write head** is then capable of detecting the magnetic charges left on the disk's surface, this is how data is read. A circuit board carefully co-ordinates rotating the disk and swinging the **actuator arm** to allow the read/write head to access any location very quickly.

SSD





The **NAND flash memory** is the primary storage medium in the SSD. It's a type of non-volatile memory that stores data as electrical charges in memory cells.

NAND flash memory chips are circuits that contain NAND flash memory. The SSD contains multiple chips, often arranged in a grid-like fashion.

The NAND flash memory is organized into **pages**. The data is read from or written to these pages. Pages are grouped together into larger units called **blocks**.

Channels are pathways through which data is transferred between the SSD controller and the NAND flash memory chips. The SSD typically has multiple channels, and the data can be stripped over them. Thus, it can be read from or written to the chips in parallel.

When it comes to reading data, random access in SSD is rather fast. But if many small reads are executed, there is a high chance that all of them are stored in the same channel.

When it comes to writing, *there are no in-place modifications*. Instead, a new page is created, and the old one is marked invalid and erased. This is because NAND flash memory cells can be written to a limited number of times before they wear out. Spreading out the write and erase operations across different memory cells helps prolong the lifespan of the SSD.

When it comes to erasing pages, *a block is a minimal-erase unit*. In other words, you can't erase individual pages within a block; you must erase the entire block. Garbage collector identifies blocks that contain invalid pages. It copies the valid data to new blocks, and erases the old blocks.

As you can see, large reads/writes are better than small ones.

23-09-14

I/O operations for secondary stodage are expensive. So the larger the single read/write operation size, the better.

Page layout

A **page** is basically a linear byte array. It begins with a fixed-size header (e.g. page number), the remaining bytes contain records. Records are stored as tuples.

There are different approaches for storing records inside a page:

- **linear page layout** - records are stored one by one, after the header. It is a good option if the size of the recod is fixed (e.g. two integers). If the record has an arbitary size (e.g. stores strings), it will be hard to replace records.
- **page layout with catalog** - uses indexes for records. Indexes are integers written after the header. They point to the corresponding records. The array of records grows from the end of the page to its center.

  The common approach in databases is to mark a record as "deleted" during the delete operation. Then garbage collector identifies valid records, copies them into a new page and erases the old page.

Table layout

A table emay contain lots of pages.

One way for storing the table is to use a **linked list**. The address of the first page is stored somewhere. Info about next pages in the list is stored in header.

Another option is to use a **page catalog**. Pages have indexes which point to a piece of secondary memory, where the page is stored. We can access any page with the same cost.

Buffer management

Once a page was read for the first time, it will probably be read again. So we can introduce a buffer layer between the DB engine and the disk.

The cache has less memory than the disk. If there is no room, we need to find a victim buffer and replace it with a new page. But not all pages can be easily replaced. The page can be marked "dirty" or be in use at the moment. Also, the DB engine process may pin a page to prevent it from eviction.

There are different cache replacement policies:

- **FIFO** (first in first out)

  Pages form a linked queue. New pages are added at the end of the queue, the victim for replacing is at the head.

- **LRU** (least recently used)

A timestamp of the last access is kept for each buffer page. The page with the smallest timestamp is the victim.

- **LFU** (least frequently used)

  Pages are placed in a circular list. There are two attributes for each buffer page:

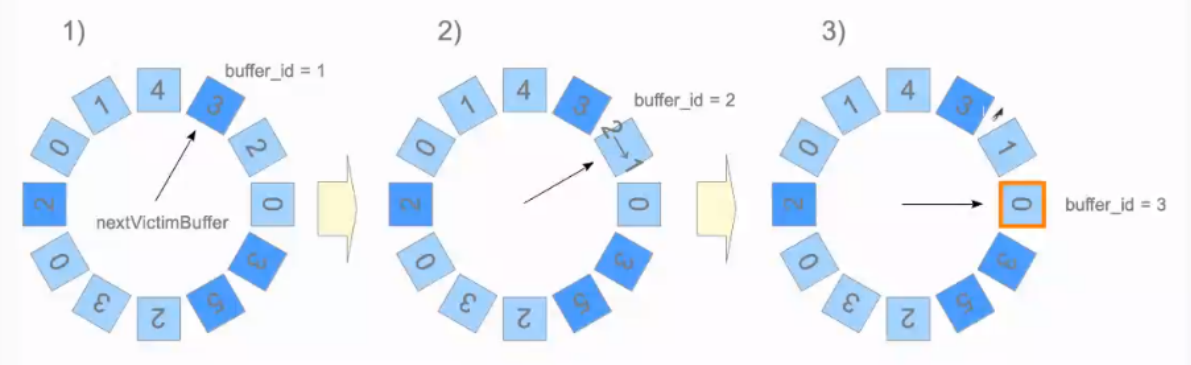
  - `pin count` - increments when DB engine process uses the page, decrements when the work is completed. If `pin count == 0` then the page is unpinned and can be removed.
  - `usage count` - increments when DB engine process uses the page, decremented by the replacement policy. A pointer scans over *unpinned* pages and decrements `usage count`. The first page with `usage count == 0` is a victim.

LFU policy: clock-sweep algorithm

Pages are placed in a circular list. A pointer goes in a clockwise direction. It sequqentially scans over pages (amount of circles can be >1) and decrements `usage count` only for unpinned pages. Once the page with `usage count == 0` found, it is a victim and will be removed.

If the page is pinned by DB engine (liked pinned forever, to prevent it from eviction), we can make `usage count` equal to $+\infty$.

In the example below, the blue pages are pinned (`pin count > 0`). Numbers in the boxes are `usage count` values.



LFU: aging algorithm

There is problem with the clock-sweep algorithm: when a lot of pages are accessed in a short period of time, their `usage count` is very high. Thus, after the work is done, pages will remain useless in the cache for a long time.

The aging algorithm accounts for both frequency and access time. It is like a combination of the clock-sweep algorithm and the LRU policy.

Every page is assoiated with a binary counter. When the page is accessed, the most significant bit in its counter is set to 1. On every clock tick [1], all counters are shifted right. The victim is a page with the smallest counter.

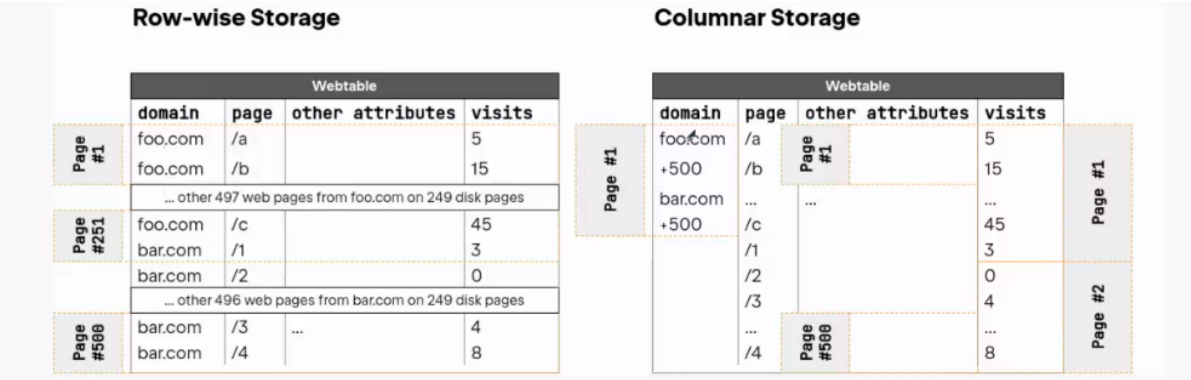
Event	Page A	Page B	Page C
READ A	1000	0000	0000
READ B	1000	1000	0000
🕘	0100	0100	0000
READ A	1100	0100	0000
🕘	0110	0010	0000
READ C	0110	0010	1000
READ B	0110	1010	1000
🕘	0011	0101	0100
🕘	0001	0010	0010
🕘	0000	0001	0001

In the example above, pages that were victims during the specific step are marked with the red dot.

23-09-21

Row-wise vs. columnar storage

There are several storage options, each designed to optimize data storage and retrieval. Row-wise and columnar storage types are used in relational databases.



Row-wise storage is where data for *each row* is stored consecutively on disk.

- suitable if you need to write and retrieve individual records frequently
- less efficient for aggregations [2]: it involves reading more data than necessary, including columns that are not part of the aggregation

In **columnar storage**, data for *each column* is stored consecutively on disk. This can be efficient for aggregations.

- highly efficient for aggregations because it allows the database to access and process only the necessary columns, reducing the amount of data read from disk

Data compression for columns

Cons:

- random access may require uncompressing $\Rightarrow$ takes more time
- Retrieving entire rows becomes costly
- Inserts and updates become more costly

Run-length encoding compression

When there are a lot of identical values in a column/row, they can be replaced by a pair `(value, occurences)`

```
foo.com, foo.com, foo.com => (3, foo.com)
```

Data encoding

For numeric columns, we can store values as a difference between the current value and the previous one. It is efficient when dealing with large values and small deltas.

value	size	stored value	stored value size
1695223026	4	1695223026	4+1
1695223030	4	4	1
1695223031	4	1	1
1695223040	4	9	1

Bitmap encoding

Applicable to columns where the count of unique values N is way less that the total count of values T.

N bitmaps are created, each with T bits, assigning one bit per row. The i_{th} bit in a bitmap is set if the i_{th} row contains the corresponding value.

```
Column values: [114, 114, 113, 114, 112, 112, 114, 113]
value=112:     [0,   0,   0,   0,   1,   1,   0,   0]
value=113:     [0,   0,   1,   0,   0,   0,   0,   1]
value=114:     [1,   1,   0,   1,   0,   0,   1,   0]
```

Bitmaps for each unique value are stored separately. Run-length compression can be used:

```
Column values: [114, 114, 113, 114, 112, 112, 114, 113]
value=112:     [(4, 0), (2, 1), (2, 0)]
value=113:     [(2, 0), (1, 1), (4, 0), (1, 1)]
value=114:     [(2, 1), (1, 0), (1, 1), (2, 0), (1, 1),
                (1, 0)]
```

In the example, compression for `value=112` helps to save some space, whereas compression for `value=114` does not.

Column family

Operations of selecting may require data from more than one column. When dealing with compressed columns, there is the need of uncompressing them. To solve these problems, **column families** are used.

A **column family** is a collection of logically related columns that are accessed and retrieved together (e.g. `name` and `surname` columns). Values of the column family are stored on the same pages.

Log-structured storage

Log files in databases are append-only files stored in the secondary storage and used for recovery purposes. When data is modified, changes are written into a log file, instead of the primary storage. In case of the system crash, the lost data can be reconstructed from the log file.

Inserts and updates are handled by the **memtable**, a tree data structure. It is implemented as an in-memory key-value store and stored in RAM (thus, fast read/write operations are ensured).

Inserts and updates are first added, as a key-value pair, into the memtable, then a log entry is created in the log file. During read operations, data is first checked in the memtable, then – if needed – in SSTables which are stored in the secondary storage.

Basically, the memtable serves as a buffer. Once the memtable reaches a certain size (or when a specific condition is met), its contents are written to the secondary storage as SSTables.

SSTables are files stored in the secondary storage. They are immutable. Once data is written to an SSTable, it cannot be modified or deleted. Instead, a new SSTable is created to reflect any updates.

Each SSTable contains a sorted collection of key-value pairs. Multiple SSTables can be periodically merged, reducing the number of SSTables and removing obsolete data. After the merge, the sorted order is still maintained in the new SSTable.

---

1. A clock tick is when a certain time interval elapses ↩

2. Aggretaions refer to operations that combine multiple data values into a single summary value. Common aggregation functions include `SUM`, `COUNT`, `AVG`, `MIN`, `MAX`. ↩