Database Internals

Database Internals
- 23-09-07
  - Org stuff
  - Recap
  - Secondary storage
    - HDD
    - SSD
- 23-09-14
  - Page layout
  - Table layout
  - Buffer management
- 23-09-21
  - Row-wise vs. columnar storage
  - Data compression for columns
  - Column family
  - Log-structured storage
- 23-09-28
  - Sorting records
    - QuickSort
    - MergeSort
      - Two-way MergeSort
      - Multiple-way MergeSort
  - Hashing records
    - Hash tables
    - Building a hash table
    - Hashing drawbacks
- 23-10-05
  - Linear hashing
    - Inserting keys
    - Table growth
- 23-10-12
  - Nested Loop Join
  - Hash Join
    - Complexity
  - Sort-Merge Join
    - Refinement \todo
  - Summary
- 23-10-19 \todo
- 23-10-26
  - What is an index
  - Sparse indexing
  - Dense indexing (secondary indexes)
  - B-trees
  - B+-trees
    - Node occupancy
    - Operations
  - Index-organized tables
    - Hash index
    - Cluster index (dublicate keys)
    - Indexes and `SELECT`

23-11-02
Voclano Framework
23-11-09

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:

```
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:

```
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:

```
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

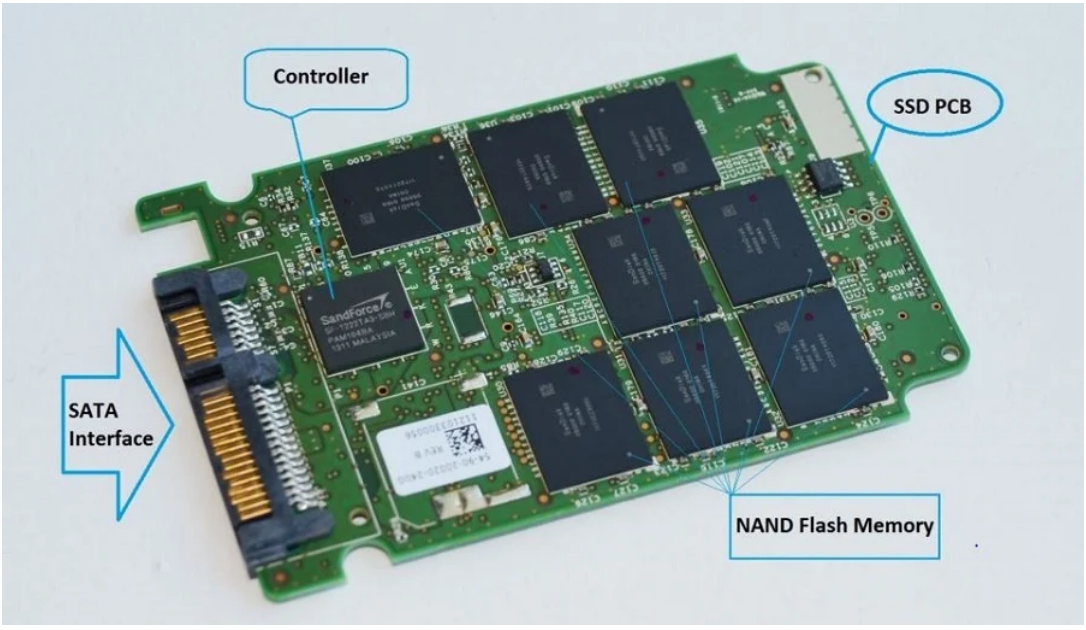
HDD

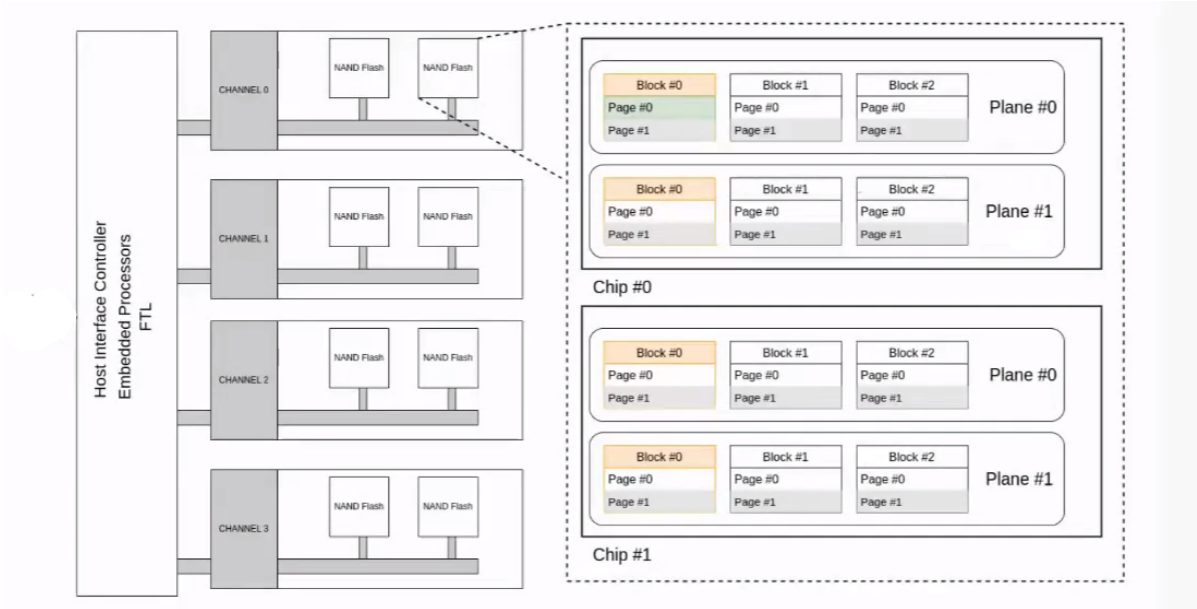
A disk consists of several *platters*. A *track* is a circle on a platter. Tracks located one above other on different platters form a *cylinder*. A *sector* is a fragment of the track.

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:

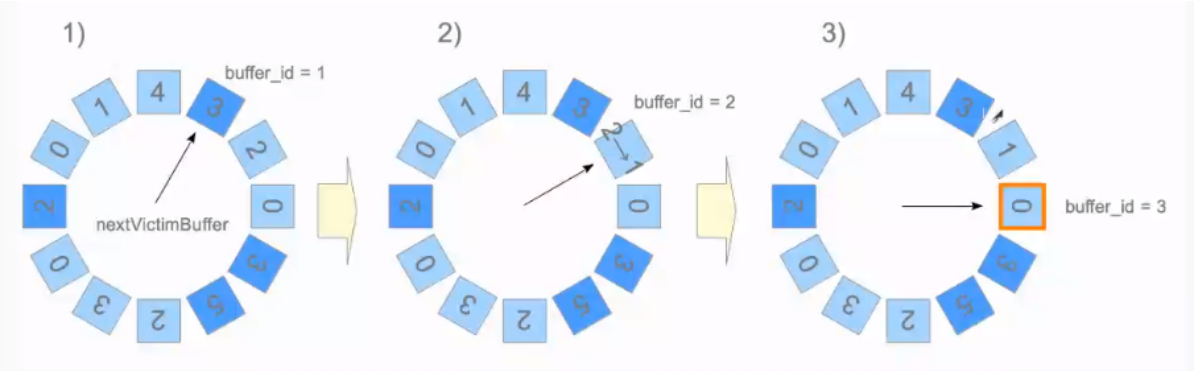
  - `pin count` - increments when DB engine process uses the page, decrements when the work is completed. If `pin count == 0` then the page is unpinned and can be removed.
  - `usage count` - increments when DB engine process uses the page, decremented by the replacement policy. A pointer scans over *unpinned*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

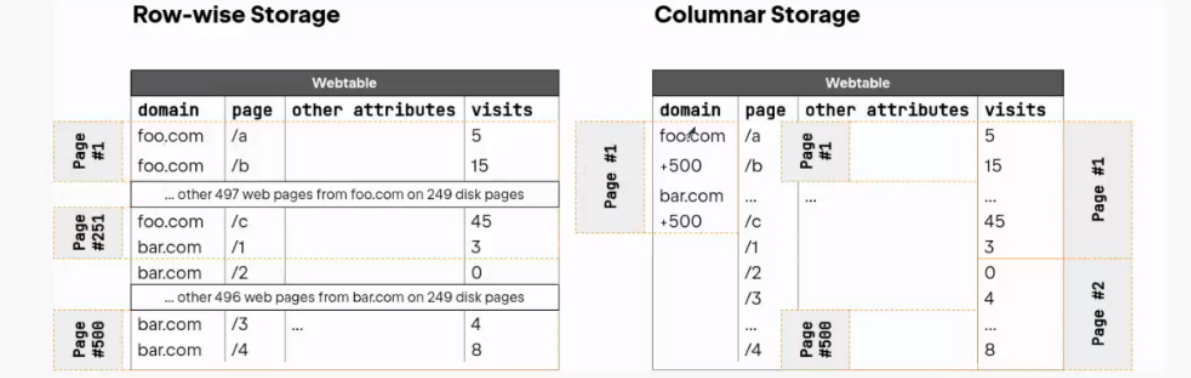
Event	Page A	Page B	Page C
READ A	1000	0000	0000
READ B	1000	1000	0000
🕘	0100	0100	0000
READ A	1100	0100	0000
🕘	0110	0010	0000
READ C	0110	0010	1000
READ B	0110	1010	1000
🕘	0011	0101	0100
🕘	0001	0010	0010
🕘	0000	0001	0001

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.

23-09-28

Sorting records

We have a table that occupies B pages and a buffer which can contain maximum M pages. $B >> M$. We want to sort the table by the value of some attribute.

When it comes to the algorithm complexity, we are going to measure it in terms of disk oerations.

QuickSort

Reminder: QuickSort involves dividing an array into two subarrays around a pivot x such that the elements in left subarray are $\leqslant x$ and elements in right subarray are bigger, then recursively sorting subarrays.

We may assume that every swap operation needs to read/write a disk page. So, the **complexity** of this approach is $O(B \log B)$ disk I/O, where B = amount of pages the table occupies.

The problem with QuickSort is that it does not account for data locality. When we swap elements, for each entry, we load a corresponding disk page into RAM, change data, then write back into the secondary memory. Elements chosen for swap can be far apart in the memory. So, in the worst scenario, we will have to load & write back twice during every swap operation.

MergeSort

Reminder: MergeSort involves recursively sorting $a[0..\frac{n}{2})$ and $a[\frac{n}{2}..n)$, then merging them.

Two-way MergeSort

Sort each page individually (using MergeSort). Then merge each pair of pages and save the result as a *sorted run*. Continue merging sorted runs until all runs are consolidated into a single result.

B = amount of pages the table occupies. In total, there are $1 + \lceil \log B \rceil$ passes, every pass divides the number or runs by 2. On each pass, the entire input is read and written, resulting in $2B$ disk I/O operations. So, the **complexity** of the two-way MergeSort is $O(2B(1 + \log B))$ disk I/O.

Example

Lets assume a database stores 9 records per page. The pages are:

```
[13, 89, 22, 58]   [56, 46, 90, 67]   [92, 88, 54, 86]
[ 8, 96, 83, 97]   [40, 11, 10, 76]   [87, 58, 64, 81]
[96, 78,  1, 77]   [94, 37, 31, 65]   [20, 12,  5, 14]
```

In the example only keys for records are written.

- Sort every page:

  ```
  [13, 22, 58, 89]   [46, 56, 67, 90]   [54, 86, 88, 92]
  [ 8, 83, 96, 97]   [10, 11, 40, 76]   [58, 64, 81, 87]
  [ 1, 77, 78, 96]   [31, 37, 65, 94]   [ 5, 12, 14, 20]
  ```

- Merge pairs of pages:

  The result of merging is saved on the disk page(s). We allocate place for the result page in the buffer, write result into it, then save it in the secondary memory.

  ```
  [13, 22, 58, 89] + [46, 56, 67, 90] => [13, 22, 46, 56]  [58, 67, 89, 90]
  [54, 86, 88, 92] + [ 8, 83, 96, 97] => [ 8, 54, 83, 86]  [88, 92, 96, 97]
  [10, 11, 40, 76] + [58, 64, 81, 87] => [10, 11, 40, 58]  [64, 76, 81, 87]
  [ 1, 77, 78, 96] + [31, 37, 65, 94] => [ 1, 31, 37, 65]  [77, 78, 94, 96]
  [ 5, 12, 14, 20] => [ 5, 12, 14, 20]
  ```

  So, we had five sorted runs. Each of them, except for the last one, consists of two pages.

- Continue merging:

```
[13, 22, 46, 56]  [58, 67, 89, 90] + [ 8, 54, 83, 86]  [88, 92, 96, 97]
   => [8, 13, 22, 46] [54, 56, 58, 67] [83, 86, 88, 89] [90, 92, 96, 97]

[10, 11, 40, 58]  [64, 76, 81, 87] + [ 1, 31, 37, 65]  [77, 78, 94, 96]
   => [1, 10, 11, 31] [37, 40, 58, 64] [65, 76, 77, 78] [81, 87, 94, 96]

[ 5, 12, 14, 20]
   => [ 5, 12, 14, 20]

...
```

Multiple-way MergeSort

The term "multiple-way" refers to the number of input sequences that are merged at each step of the merging process.

In the multiple-way MergeSort, during the **sorting phase**:

1. Divide the source array into segments, each containing M or fewer than M pages (M is the cache size).
2. Sort each segment in memory and save the resulting sorted run to the disk.

During the **merging phase**:

1. Allocate one output page in the cache.
2. Load heads (the first pages) of the sorted runs into the cache.
3. Merge them using a multiple-way MergeSort. When one of the heads get exhausted, pull the next page from the corresponding sorted run.
4. Write the result into the output page and flush it into the secondary memory when it fills up.

B is the total amount of pages. In the sorting phase, the entire input is read, sorted, then written, resulting in $2B$ disk I/O operations. In the merging phase, we read from all sorted runs and write into the output, this again results in $2B$ disk I/O operations. So, in total, the **complexity** of the multiple-way MergeSort is $4B$ disk I/O.

The estimation is true if a **specific condition** is satisfied: the amount of sorted runs is $\leqslant M$ (M is the cache size). Otherwise, not all heads of sorted runs will fit into the cache.

If we have M segments, $\frac{B}{M}$ is the length of each segment. We want the segment's sorting phase to be linear, so $\frac{B}{M} \leqslant M \Rightarrow$ the algorithm is linear if $B \leqslant M^2$.

In general:

- Sorting produces $\lceil \frac{B}{M} \rceil$ sorted runs, length of each run is $\leqslant M$.
- When merging, we divide the number of sorted runs by M. Groups are merged independently, then the process repeats until there are $\leqslant M$ sorted runs left. Then the final merging phase starts.
- In total, we need 1 sorting pass and $\log_M \lceil \frac{B}{M} \rceil$ merging passes. On every pass, we read and write each page. The resulting **complexity** is $2B(1 + \log_M \lceil \frac{B}{M} \rceil)$.

Exmaple

$M = 4$. The pages we are going to use are:

```
[13, 89, 22, 58]   [56, 46, 90, 67]   [92, 88, 54, 86]
[ 8, 96, 83, 97]   [40, 11, 10, 76]   [87, 58, 64, 81]
[96, 78,  1, 77]   [94, 37, 31, 65]   [20, 12,  5, 14]
[ 6, 84, 90, 96]   [16, 42, 45, 52]
```

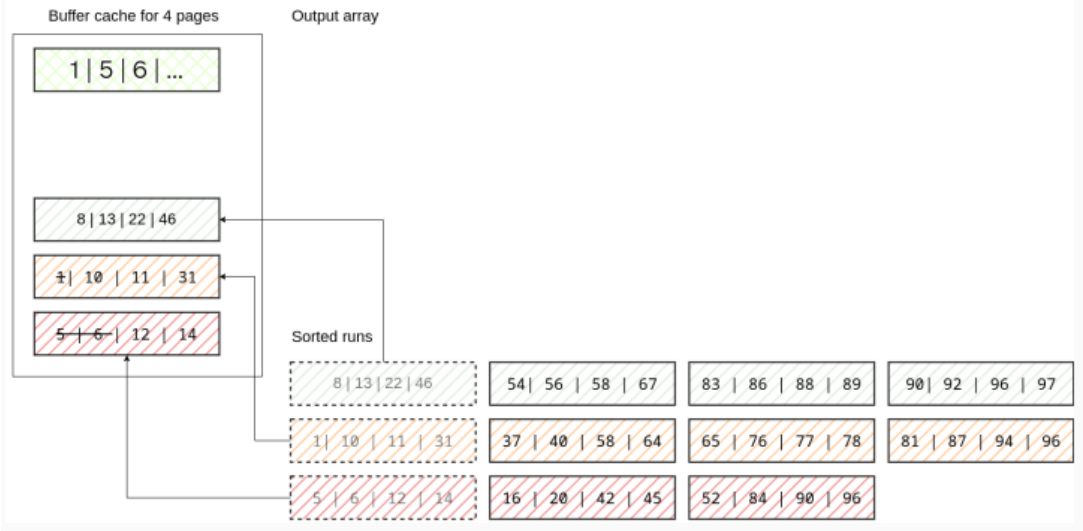
- The sorting phase:

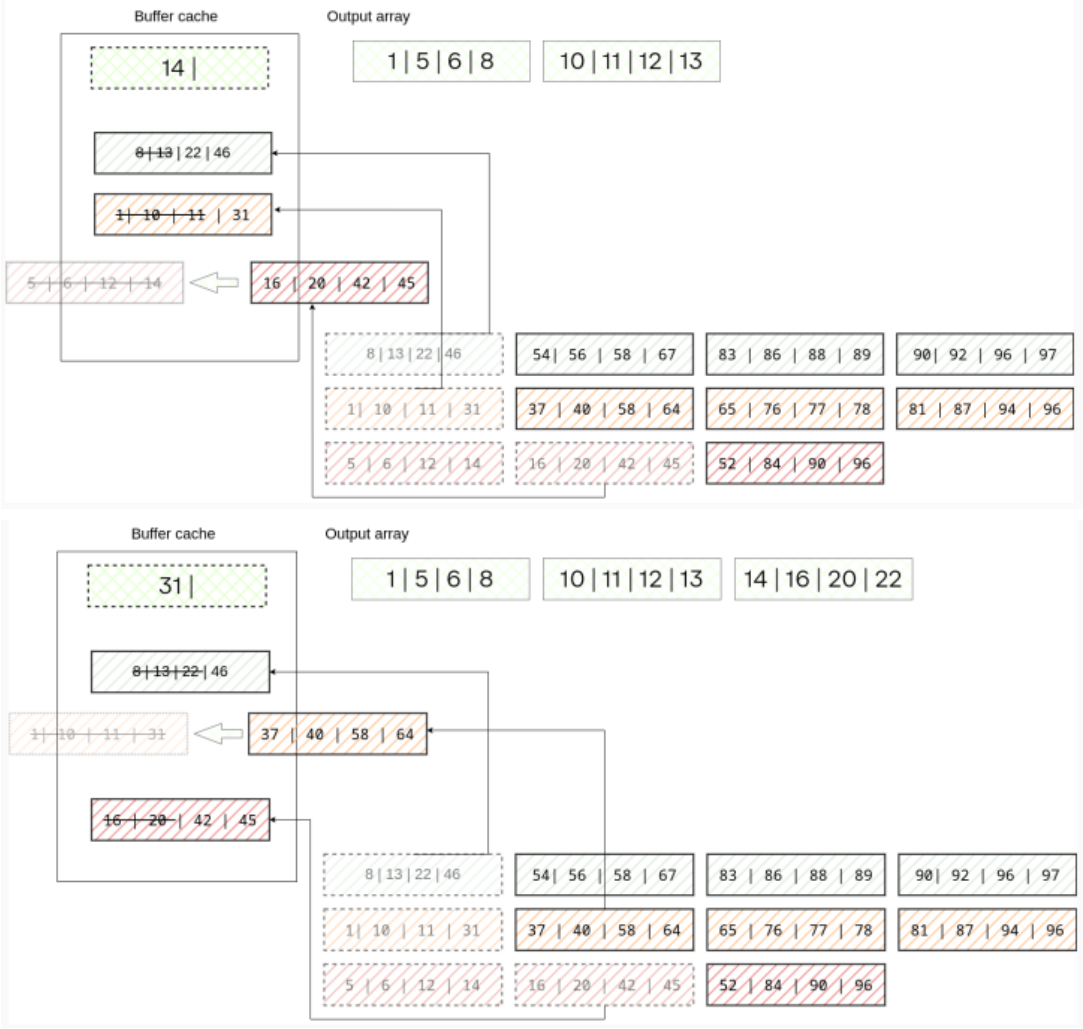
```
[13, 89, 22, 58] [56, 46, 90, 67] [92, 88, 54, 86] [ 8, 96, 83, 97]
   => [8, 13, 22, 46] [54, 56, 58, 67] [83, 86, 88, 89] [98, 92, 96,
97]

[40, 11, 10, 76] [87, 58, 64, 81] [96, 78,  1, 77] [94, 37, 31, 65]
   => [1, 10, 11, 31] [37, 48, 58, 64] [65, 76, 77, 78] [81, 87, 94,
96]

[20, 12, 5, 14] [6, 84, 90, 96] [16, 42, 45, 52]
   =>  [5, 6, 12, 14] [16, 20, 42, 45] [52, 84, 90, 96]
```

- The merging phase:





etc.

Hashing records

Hash tables

Hash table is a data structure that partitions a set of keys into M buckets. Buckets are identified by the index. A **hash function** $h(k)$ maps keys to integer values (bucket indices). If we want to insert a value in the hash table, we compute its hash value and insert it in the bucket at an index equal to the hash value (modulo the size of the array).

Strategies for resolving collisions:

- **Chaining:** store a list of keys in the dame bucket (linked list or binary tree, consisting of key-value pairs).
- **Open addressing:** when a collision occurs, the hash table looks for the next available bucket in a sequence.
- **Dynamic Hashing Policies:** when a collision occurs, and the current hash table becomes too crowded, new buckets are created and hash values for the old data are reevaluated.

The **load factor** of a hash table is a real number $l \in (0, 1)$ that tells how full the hash table is.

$$l = \frac{\#elements}{\#buckets}$$

Most hash tables have a **max load factor** α, a constant number between 0 and 1 that is an upper limit for the load factor. A common value for the max load factor is 0.75.

When we insert a new value to the hash table, we calculate the new load factor l. If $l > \alpha$, then we increase the number of buckets in the hash table (usually by creating a new array of twice as many buckets), then recalculate buckets for old values.

Building a hash table

In databases, a bucket element is a disk page. A disk page may contain multiple hash keys and their associated values.

Chaining is ussually used to resolve colisions (index of the next page in the linked list is written in the header of the current page).

We have a table that occupies B disk pages and a buffer that accomodates up to $M + 1$ pages. $B >> M$. We want to hash the table records by the value of some attribute.

- 1 buffer slot is used for reading input
- Other M buffer slots are used for hashing, one slot represents one bucket.
- Read the input, apply the hash function $h(k) \mod M$, place the recond into the appropriate buffer page. If some buffer page fills up, flush it to the disk.

Hashing drawbacks

- Poor selection of a hash function can result in uneven bucket sizes. It impacts the performance, sometimes we will have to scan for more data in a particular bucket.
- Non-uniform data distribution among buckets may lead to uneven bucket sizes. The same problem with the performance as stated above (good news: databases check distribution of keys and do search for an efficient algorithm to hash keys).

23-10-05

Linear hashing

The last i bits are taken from the result of the hash function. There are $n < 2^i$ buckets in the table.

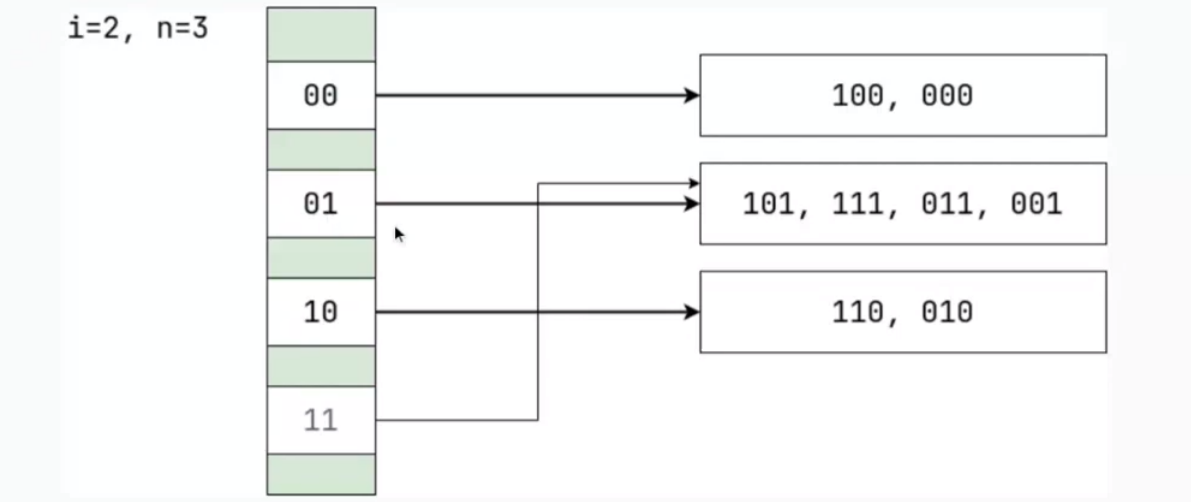
Inserting keys

We evaluate $h(k)$, we take the last i bits. Assume the value of $\overline{x_{i-1}\dots x_0}$ in equal to m.

- If $m < n$, the m-th bucket exists and we put the key into it.
- If $n \leqslant m < 2^i$, then the bucket with number $m - 2^{i-1}$ exists, we put the key there.

  In simple words, we "turn off" the largest bit and choose the bucket using the updated value.

Example



The bucket number 11 is virtual.

$h(k) = \dots 100,\ m = 0 < n \ \Rightarrow\ \texttt{bucketNumber} = 0.$

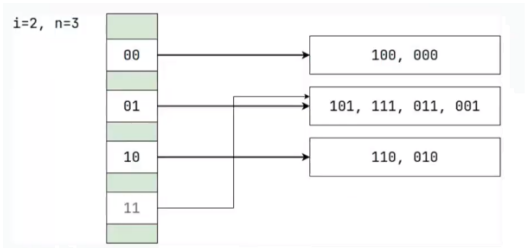
$h(k) = \dots 111,\ m = 3,\ n \leqslant 3 < 2^i \ \Rightarrow\ \texttt{bucketNumber} = 3 - 2 = 1.$

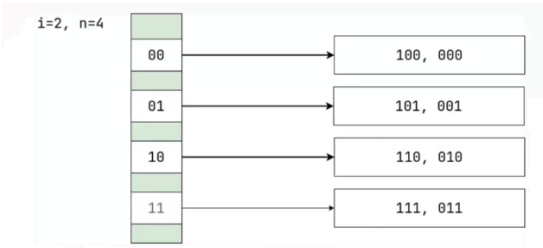
Table growth

The hash table should be loaded for not more than 80%. If there are k keys and n buckets, we want the amount of records to be $\leqslant r = 0.8nk$.

- If the amount of keys is larger than r, we create a new bucket with the number $n = \overline{0b_{i-1}\dots b_0}$. Values are moved from the bucket $n - 2^{i-1}$ into the bucket n.

  $n \to n + 1$.

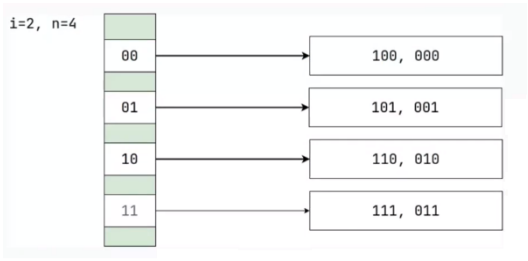


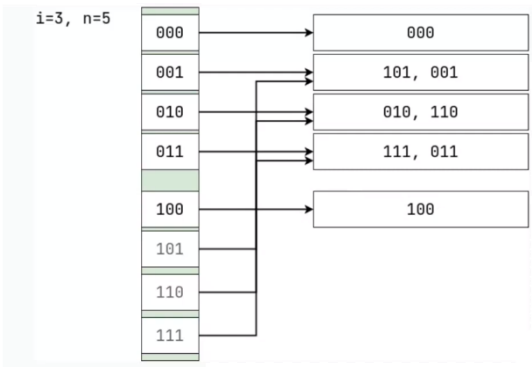


- If $n \geqslant 2^i$, then $i \to i + 1$.

  The number of buckets becomes twice bigger. All of added buckets are virtual, except for one.

  Buckets for the old data are reevaluated.





23-10-12

Nested Loop Join

SNLJ is a literal implementation of the inner join definition:

```
def simple_nlj(table1, table2, out, join_predicate):
  for r1 in table1.records:
    for r2 in table2.records:
      if join_predicate(r1, r2):
        out.append(r1 + r2)
```

R is the outer table, S is the inner one.

Complexity is $B(R) + T(R) \cdot B(S)$ where $B(.)$ is the scan of the whole table and $T(.)$ is a tuple.

Consider all pairs of pages from the input tables and execute SNLJ for each pair:

```
def page_nlj(table1, table2, out, join_predicate):
  for p1 in table1.pages:
    for p2 in table2.pages:
      simple_nlj(p1, p2, out, join_predicate)
```

```
def block_nlj(table1, table2, out, join_predicate):
  repeat(ceil(len(table1.pages) / cache.size)) {
    page_nlj(
      read_next_chunk(table1, cache.size),
      table2, out, join_predicate
    )
  }
```

Hash Join

Naive Hash Join

- Hash the records of the *build table*, say *R*, in memory using the join attribute as a hash key, and allocating *M* buffers for the hash buckets.
- Scan the other *probe* table *S* using 1 buffer. Apply the same hash function to its join attribute, search for matches in the appropriate bucket, and output the matched pairs.

Naive Hash Join works if the size of each bucket of the build table is $\leqslant M$ where M is the size of the cache (so, the size of the build table should be $\leqslant M^2$).

Grace Hash Join I

- What if neither of the tables can fit in the cache?
- Hash both tables by the join attribute value using the same number of buckets, writing the bucket pages to the temporary disk storage.
- Potentially matching rows will end up in buckets with the same numbers.

/todo insert Grace Hash Join II

Complexity

/todo

Sort-Merge Join

`equi` stends for equality.

Sort-Merge Equi-Join

- Suppose that we need to execute and equi-join.
- Sort both tables by the join attribute values.
- Join procedure is similar to merge:
  - Load the sorted table heads into the cache.
  - Advance the table with the least value of the join attribute until a match with the other table is found.
  - Upon reaching the matching rows, join the runs of matching records using Nested Loop Join.
  - If we're lucky, at least one of the matching runs will fit into memory and we will join them in linear time.

Sort-Merge Join Complexity

- The cost is the cost of sorting + merge-join:

$$4 \times (B(R) + B(S)) + (B(R) + B(S))$$

provided that $B(R) < M^2, B(S) < M^2$ and the runs of records with the same value of join attribute are short enough to fit entirely in memory.

Refinement \todo

min 46-47

Summary

- We have a choice of the join algorithms.
- Nested Loop Join works fine for small joins.
- Hash Join is perfect for equi-joins of large tables.
- Sort-Merge Join works better for large joins if join predicate is not an equality or if one of the tables is already sorted.
- The decision on which algorithm to use is made by the database engine during query execution planning.

23-10-19 \todo

23-10-26

What is an index

Index is a persistent data structure that may speed up some operations.

- redundant: using indexes is not compulsory, we can search without them
- persistent: indexes are not built for every query from scratch but stored separately in the memory

We build an index for the values of a table column. The indexed column values are **index keys**. We need to associate index keys with the appropriate table records (or table pages).

In the context of indexing, we **distinguish between data pages**, where the actual table records are stored, **and index pages**, where index data structure is maintained.

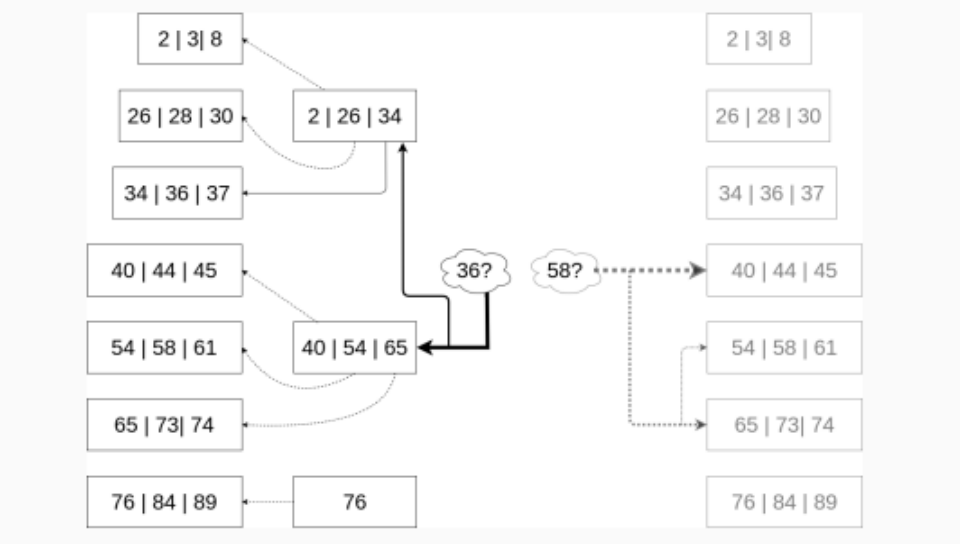
Indexes must remain consistent, meaning they need to be updated whenever there are inserts, updates, or deletions in the table. Index maintenance may be quite expensive, particularly when the table experiences frequent updates.

Sparse indexing

Extract the first value from each page and create a mapping with ”key - page pointer” pairs. This mapping is referred to as a **sparse index**. Records in the sparse index consist of the search key and page pointer, and occupy less space than the table records. It makes using sparse indexing compact and efficient.

Also, we can create a **second layer** of the sparse index on top of the first, making the operations less time-consuming.

Example (with one and two layers):



Here the *sparse* indexing is shown: indexes are assigned only to the first pages in each cluster. *Dense* indexing means indexes are assigned to each record in the table.

Dense indexing (secondary indexes)

What if the table is sorted by `id` while we want to search by some other attribute, is e.g. by `name`?

We cannot reorder the table, or make a copy ordered by name, since there is a lot of data in the table. But we can build **dense index** where mappings "key - pointer to table record" are stored.

Dense index now can be sorted. We can also create a second layer (which will be sparse).

Dense indexes are called **secondary indexes**. We may have as many secondary indexes as we need.

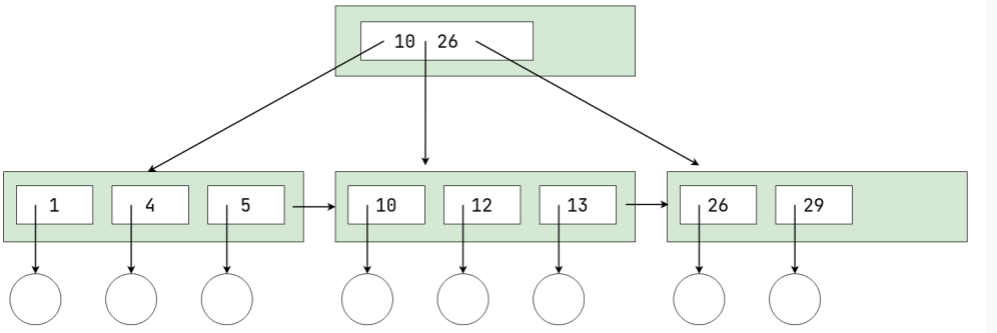
B-trees

- 'b' stands for nobody knows what. It was developed 50 years ago, by Rudolf **B**ayer and Edward M. McCreight. Both or them were working at the **B**oeing Research Labs. B-tree can be **b**alanced or **b**inary. Data stored in nodes can be considered as a **b**lock of pages. A lot of letters 'b', but it is a mystery what 'b' stands for.
- Index layers:
    - dense leaf layer
    - sparse second layer on top
    - until we have a 1-page layer
- The resulting structure is a search tree with a large number of branches. Two adjacent nodes define a search range.
- We will distinguish between **leaf nodes**, **inner nodes**, and the **root node**.

B+-trees

B+-tree is a variation of B-tree. In a B+-tree, only the leaf nodes contain the actual data, while the inner nodes contain keys but no data. This design difference allows for faster searches in practice.

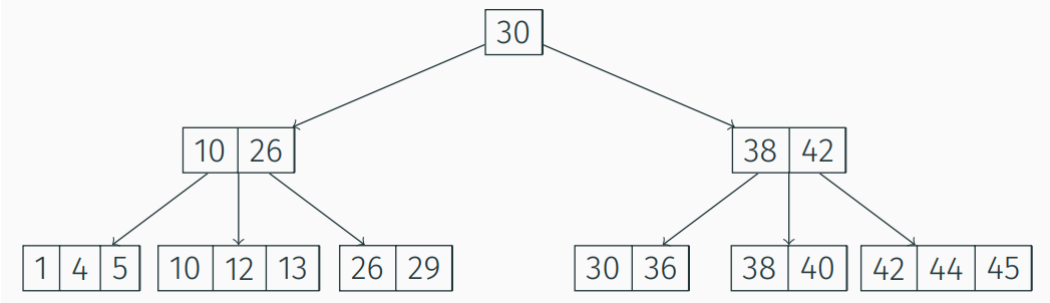
Suppose we've got N index keys in total. A disk page can accommodate up to m key-pointer pairs. Each index page will hold $k \leqslant m$ keys and $k + 1$ pointers to other pages.



Leaf nodes form a dense index over the indexed attribute. k pointers are parts of key-pointer pairs, and one more pointer is used to create a linked list from nodes that are on the same level. Thus, we can iterate over them without refering to other parts of the tree.

Inner nodes form a sparse index for leaves. In each inner node, there are k keys and $k + 1$ pointers which point to $k + 1$ substrees. Keys and pointers are alternated, like in the picture. The pointer between keys k_i and k_{i+1} references to a node with a range $[k_i, k_{i+1})$. For example, the pointer before 10 leads to $[1, 10)$. The pointer after 26 leads to $[26, 29)$.

Root node stores one key. Its value is equal to the smallest one from the right subtree. Also, its value is not used in the second layer.

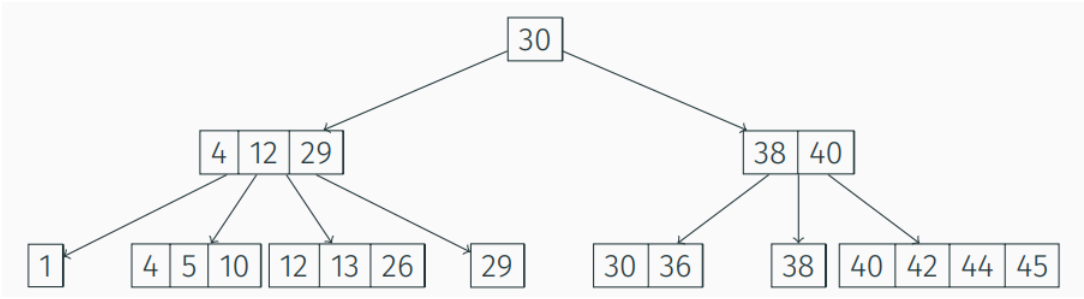


Node occupancy

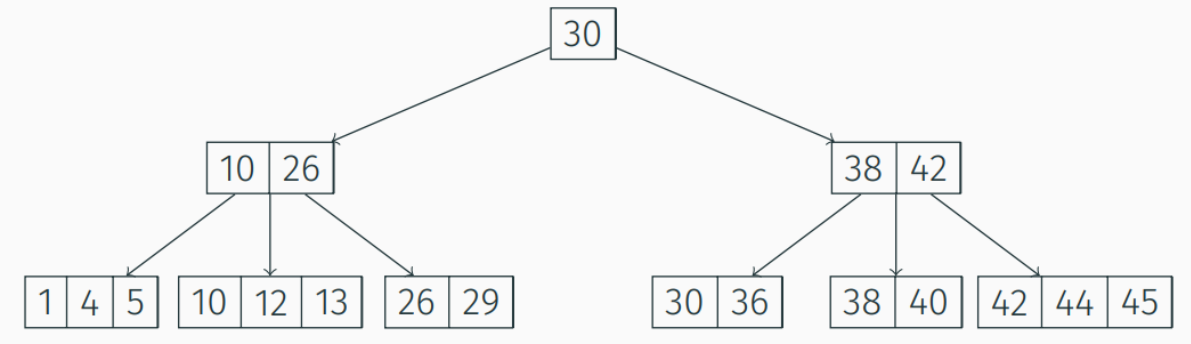
- There are not less than $\lfloor \frac{m+1}{2} \rfloor$ pointers to table records in leaf nodes.
- There are not less than $\lceil \frac{m+1}{2} \rceil$ pointers referencing to nodes of the lower layer.
- At least two pointers are used in the root node.

Examples

Suppose that $m = 3$, we can hold up to 3 keys and 4 pointers. Leaf nodes must have at least 2 data pointers, inner nodes must have at least 2 pointers to the lower layers.



The example above is not correct. Some leaves store only one data pointer. Also, $m = 3$ means not more than 3 keys can be written into an index page, which is false for the last leaf node.



The example above is correct.

Operations

Search

Explore the key ranges starting from the root, follow the pointer that sits in the appropriate range.

Complexity is $\log_m(N)$ where N is a total number of keys and m is a maximum number of keys one index page can store.

Insert

- Find a leaf where a new key is supposed to be.
- If there is enough space in the leaf, insert a new key-pointer pair.
- If the leaf page is full, we need to split it (see below).

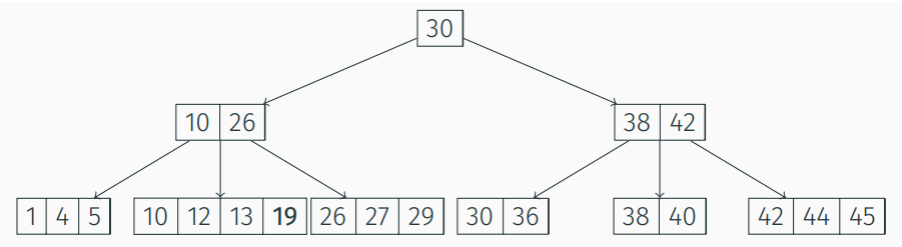
Complexity is $O(\log_m N)$.

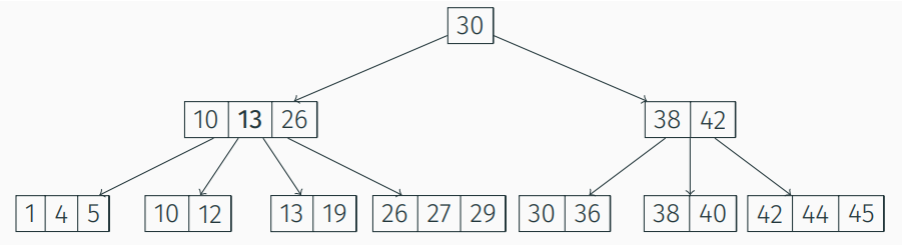
Split for leaf node

We inserted a key-pointer value into a leaf node `l`, and the amount of pairs in now $m+1$. Then, the first $\lceil \frac{m+1}{2} \rceil$ pairs remain where they are. A new leaf node `new_l` is created, and the last $\lfloor \frac{m+1}{2} \rfloor$ pairs are relocated into it.

The first key from `new_l` is going to be inserted into the parent of `l`. If there is space for it, we insert a new key and pointer referencing `new_l`. Otherwise, see the split for the inner node.

For example, $m = 3$, we insert key 19 and need to split a leaf node.

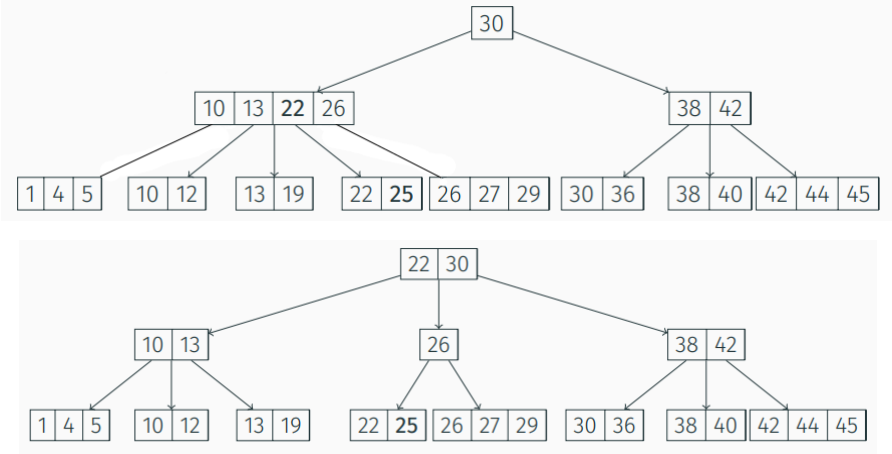




Split for inner node

- We have the inner node `i` where there are $m+1$ keys and $m+2$ pointers.
- First $\lceil \frac{m+2}{2} \rceil$ pointers and $\lceil \frac{m}{2} \rceil$ keys remain where they are.
- Last $\lfloor \frac{m+2}{2} \rfloor$ pointers and $\lfloor \frac{m}{2} \rfloor$ keys are moved into a new inner node `new_i`.
- One key in the middle is left untouched. It is moved to the parent of `i`. Also, a new pointer in the parent of `i` is created, referencing to the node `new_i`.

For example, $m = 3$, we insert key 25 and, as a result, there are 4 keys in one inner node.



Remove

Возможно, не стоит удалять ключи, а просто использовать флажок. Если удалять, то важно проверять инварианты и, при необходимости, менять всё дерево. Подробно на удалении мы останавливаться не будем.

Index-organized tables

In certain database engines, table records are *heap-organized*, where new records are placed wherever free space is available, and the records are not ordered in any particular way.

In contrast, an *index-organized* table stores records directly within the index structure. The index key is typically the table's primary key or an internal record identifier. The key is associated with the record directly (not with the page).

The drawback of indexing is records may change their address, when the index structure grows and some pages have to be splitted. This may trigger rebuilding of other indexes.

Hash index

The idea is to hash table records, then use index attribute as the hash key and pointers to the data pages as the values.

The complexity of operations is $O(1)$ on average, but **with some conditions**: the size of the bucket should have a lower bound. Usually, the amount of buckets is doubled when needed, involving rehashing all keys and rebuilding the whole table which are expensive processes. We want to modify only one bucket and leave other buckets as they are. The example of this approach is linear hashing.

Cluster index (dublicate keys)

When the indexed column of an index-organized table permits duplicating index keys, we describe the index as **clustered** or **clustering**.

A clustering index organizes the indexed table rows so that all rows with the same index key value are stored next to each other, packed to a minimum possible number of pages.

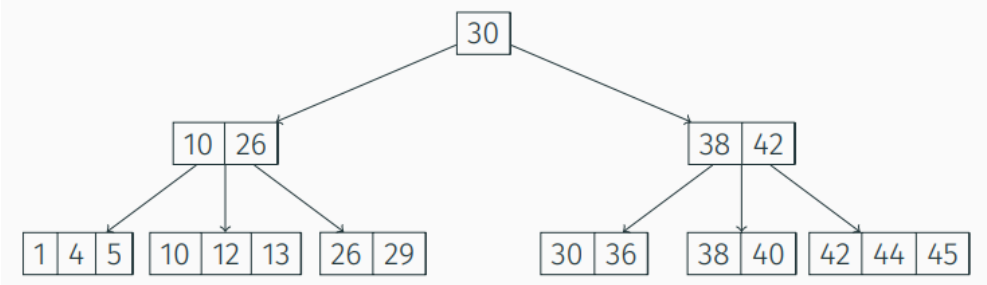
If we allow for storing duplicate keys in the leaves, we need to modify the meaning of keys in the inner nodes:

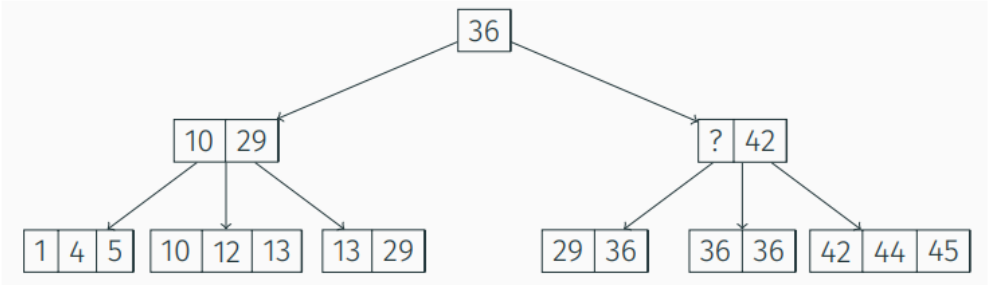
- *New key* is the key which value occurs at least once in the right subtree but never occurs in the left subtree

  In inner node, there are k keys and (k+1) pointers.
  Thus, for each key, there is a left and a right subtree.

- If we have keys $k_1, \ldots, k_n$ in the inner node then k_i is the smallest *new* key that appears in $(i+1)$-th subtree (the one with boundaries $[k_i, k_{i+1})$).
- Some inner nodes may have `null` values

Example 1: replace 26 with 13, 30 with 29, and {38, 40} with 36





29 is in the inner node now, since it occurs in the right subtree and does not occur in the left one.

The root is not 29 but 36, since 29 is in the left subtree and 36 is not.

There is a `null` value instead of 38, since 36 occurs in both left and right subtrees.

Example 2: look for keys when value in some inner node is `null`

If the value is `null`, it means all keys from the right subtree can be also found in the left subtree. So, when looking for entries with a specific key, we should start from the left subtree.

For example, we are looking for entries with the key 36. From the root we go to the right subtree. The value in the inner node is `null` there. It means 36 is in both left and right subtrees of that inner node.

Now lets look for entries with key 29. From the root we go to the left subtree, then to the right. In the end, we get both entries, since leaf nodes have sibling pointers to construct a linked list.

Indexes and `SELECT`

Notations:

- $R(A, B)$ is a table consisting of two columns A and B
- $B(R)$ is the number of pages holding the records of R
- $T(R)$ is the number of rows in R
- $V_A(R) = T(\pi_A(R))$ is cardinality (мощность) of column A -- the total number of distinct values stored in column A. The value is between 1 and $T(R)$.

Suppose we have an index built for column A. We need to execute a query that looks like

```
SELECT FROM R WHERE A = ....
```

Without using the index, we scan through all pages which takes $B(R)$ I/O.

No suppose we use the index.

If the index is clustering, rows with the same value are packed to the min possible amount of pages. If the distribution of values is uniform, we need $\lceil \frac{B(R)}{V_A(R)} \rceil$ I/O.

If the index is not clustering, the worst case is when every row that matches the search condition is stored in its own page. Then we need $\lceil \frac{T(R)}{V_A(R)} \rceil$ I/O.

EXAMPLES

- Let $B(R(a,b)) = 1000, T(R) = 20\,000$, 20 records/page. How many I/O will cost the query `SELECT FROM R WHERE a=42`
- If there is no index then 1000 I/O
- If $V_a(R) = 20$ and there is a clustering index then $\lceil \frac{1000}{20} \rceil = 50$ I/O.
- If $V_a(R) = 20$ and there is a non-clustering index then $\frac{20000}{20} = 1000$ I/O
- If $V_a(R) = 10$ and there is a non-clustering index then $\frac{20000}{10} = 2000$ I/O
- If $V_a(R) = 20\,000$ and there is a non-clustering index then $\frac{20000}{20000} = 1$ I/O

23-11-02

Voclano Framework

Volcano is a theoretical framework for building efficient query processors. Some key aspects:

- Operator-Based Processing: query execution is a tree of operators. Operators can be selection/filtering, joins, projection, etc., each performing some specific operation on data.
- Pipelined Processing: data flows through the operators in a pipeline fashion. An operator output is fed to the parent operator in a query plan.
- Lazy materialization: operator output is calculated lazily if possible.
- For pipelined processing and laziness iterator-like interface is used: `open()`, `next()`, `close()`.

  `open()` prepares the operation and opens iterators down to the plan leaves at least in one branch of the query tree.

  `close()` releases the acquired resources and closes all iterators in the subtree.

23-11-09

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`. ↩