Introduction to Databases

Gianluca Quercini

Laboratoire de Recherche en Informatique
CentraleSupélec

2019 – 2020

CentraleSupélec

Indexing in Relational Databases

Indexing

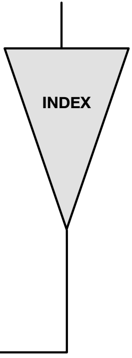
- **Indexing.** Mechanism to **speed up** data access.

Find the employee 8

Without indexing

With indexing

Employee					
codeE	**first**	**last**	**position**	**salary**	**codeD**
1	Joseph	Bennet	Office assistant	55,000	14
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
4	Mary	Green	Credit analyst	65,000	62
5	William	Russel	Guidance counselour	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Assistant director	120,000	14



Indexing

- An **index** is a **data structured** stored in the DBMS as a **file**.
- An **index file** is relative to a **column** or a **set of columns** of a table.
- An index file is a sequence of **records** (a.k.a., **index entries**).
  - Each record is a pair (`search_key`, `pointer`).
  - `search_key`: values used to search in a table (e.g., *codeE*).
  - `pointer`: reference to the row in the table corresponding to the search key.
- **Primary index.** The search key is the **primary key**.
- **Secondary index.** The search key is another column.
- Two types of indexes: **ordered** and **hash**.
- **Ordered indexes.** The index file is **sorted** by search key.
  - **Linear** indexes: the index file is a sequence of **key-value pairs**.
  - **Tree-based** indexes: the index file stores a **tree**.
- **Hash indexes.** The index file is a **hash table**.

Linear Indexes

- The search keys are **sorted** in the index file.

Index on codeE	
1	
2	
3	
4	
5	
6	
7	
8	

Employee					
codeE	**first**	**last**	**position**	**salary**	**codeD**
4	Mary	Green	Credit analyst	65,000	62
5	William	Fisher	Guidance counselour	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
1	Joseph	Bennet	Office assistant	55,000	14
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Assistant director	120,000	14

Linear Indexes – Primary index

- In this example, we have a **primary index**.
- The search key is the **primary key**.

Index on codeE	
1	
2	
3	
4	
5	
6	
7	
8	

Employee					
codeE	**first**	**last**	**position**	**salary**	**codeD**
4	Mary	Green	Credit analyst	65,000	62
5	William	Fisher	Guidance counselour	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
1	Joseph	Bennet	Office assistant	55,000	14
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Assistant director	120,000	14

Linear Indexes – Secondary Index

Index on last	
Bennet	
Doe	
Fisher	
Green	
Smith	
Watson	
Young	

Index on codeE	
1	
2	
3	
4	
5	
6	
7	
8	

Employee					
codeE	**first**	**last**	**position**	**salary**	**codeD**
4	Mary	Green	Credit analyst	65,000	62
5	William	Fisher	Guidance counselour	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
1	Joseph	Bennet	Office assistant	55,000	14
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Assistant director	120,000	14

Linear Indexes – Secondary Index

Index on last	
Bennet	
Doe	
Fisher	
Green	
Smith	
Watson	
Young	

Index on codeE	
1	
2	
3	
4	
5	
6	
7	
8	

Employee					
codeE	**first**	**last**	**position**	**salary**	**codeD**
4	Mary	Green	Credit analyst	65,000	62
5	William	Fisher	Guidance counselour	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
1	Joseph	Bennet	Office assistant	55,000	14
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Assistant director	120,000	14

- Two levels of indexes. One possible solution is to have a **clustered index**.

Linear Indexes – Clustered Index

- When the index is **clustered**, the records of the table are **physically sorted** on disk.
- Only one clustered index in a table.
- Many non-clustered indexes are possible.

Index on last	
Bennet	
Doe	
Fisher	
Green	
Smith	
Watson	
Young	

Employee					
codeE	**first**	**last**	**position**	**salary**	**codeD**
1	Joseph	Bennet	Office assistant	55,000	14
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
4	Mary	Green	Credit analyst	65,000	62
5	William	Fisher	Guidance counselour	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Assistant director	120,000	14

Linear Indexes – Composite Indexes

- Index on several columns (e.g., last and first name).
- **Composite clustered index**. The rows are sorted by the last name and then by the first name.

Employee					
codeE	**first**	**last**	**position**	**salary**	**codeD**
1	Joseph	Bennet	Office assistant	55,000	14
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
5	William	Fisher	Guidance counselour	35,000	25
4	Mary	Green	Credit analyst	65,000	62
6	Elizabeth	Smith	Accountant	45,000	62
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Assistant director	120,000	14

Linear Indexes – Composite Indexes

- Index on several columns (e.g., last and first name).
- **Composite non-clustered index**. The search key is a composition of the last and first name. The search keys are sorted.

Index on LastFirst	
BennetJoseph	
DoeJohn	
FisherPatricia	
FisherWilliam	
GreenMary	
SmithElizabeth	
WatsonMichael	
YoungJennifer	

Employee					
codeE	**first**	**last**	**position**	**salary**	**codeD**
1	Joseph	Bennet	Office assistant	55,000	14
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
4	Mary	Green	Credit analyst	65,000	62
5	William	Fisher	Guidance counselour	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Assistant director	120,000	14

Linear Indexes – Composite Indexes

- The **order** of the columns in a composite index matters.
- The query SELECT * FROM Employee WHERE last='Smith' **will** benefit from the index (first column of index).
- The query SELECT * FROM Employee WHERE first='Mary' **will not** benefit from the index (second column of the index).

Index on LastFirst	
BennetJoseph	
DoeJohn	
FisherPatricia	
FisherWilliam	
GreenMary	
SmithElizabeth	
WatsonMichael	
YoungJennifer	

Employee					
codeE	**first**	**last**	**position**	**salary**	**codeD**
1	Joseph	Bennet	Office assistant	55,000	14
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
4	Mary	Green	Credit analyst	65,000	62
5	William	Fisher	Guidance counselour	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Assistant director	120,000	14

Linear Indexes – Pros and Cons

Advantages:

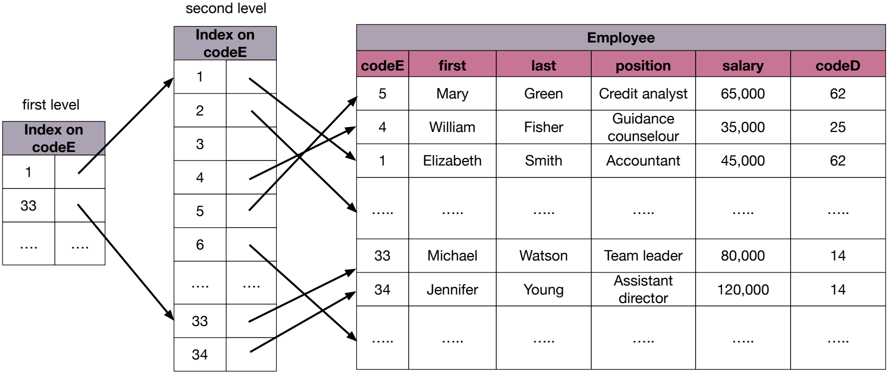
- **Efficient search**: $O(\log n)$ (binary search).
- **Range queries** are supported.

Disadvantages

- **Update high cost**: $O(n)$.
- The search is efficient when the index can be loaded **entirely** in main memory.
  - Otherwise, the search performance is degraded by disk accesses.
- Depending on the size of the index, this might not be possible.
- One possible solution is to have a **multi-level index**.

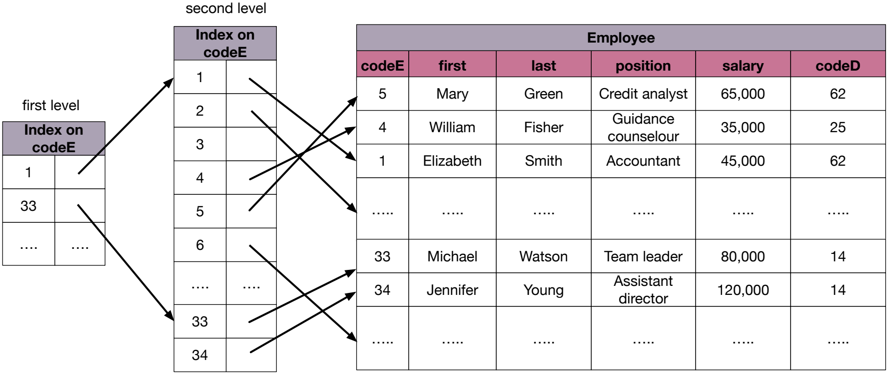
Linear Indexes – Multi-level Index

- First level can be in memory.
- Only the portion of interest of the second level can be loaded into memory.



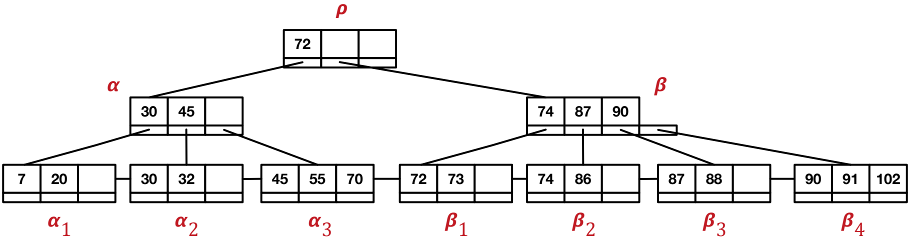
Linear Indexes – Multi-level Index

- How many levels?
- No clear definition of *level*.



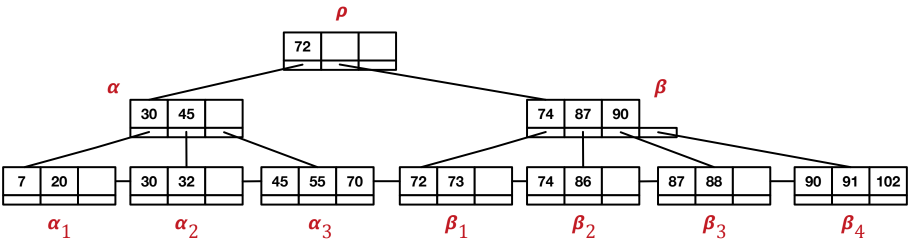
B+ Trees

- **Leaves** store the search keys.
- **Internal nodes** store **values** that guide the search.
- The tree is **height balanced**. All leaves are at the same distance from the root.



B+ Trees

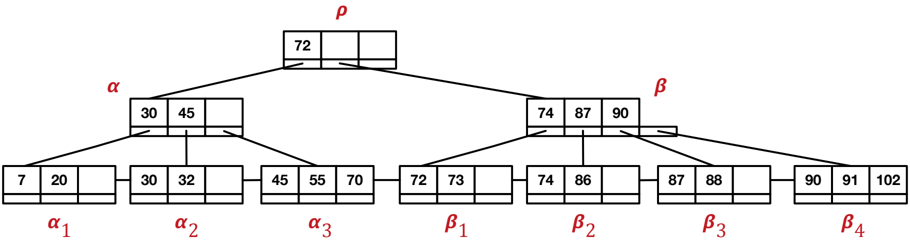
- **Branching factor (b).** Maximum number of children of a node.
- Root has either zero or at least two children.
- Each internal node has between $\lfloor b/2 \rfloor$ and b children.
- If an internal node has m children, it has $m - 1$ values.
- Each leaf has between $\lfloor b/2 \rfloor$ and $b - 1$ search keys.
- The leaves are concatenated in a doubly linked list.



B+ Trees

Searching for the search key 88.

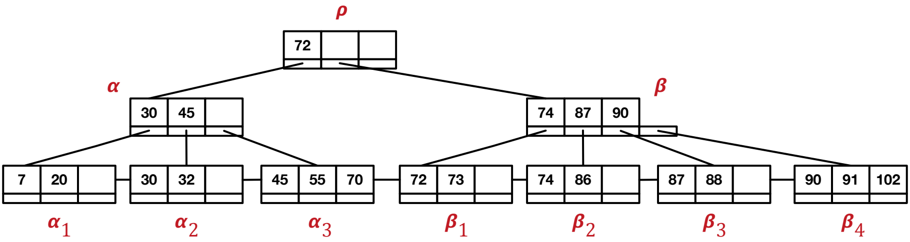
- $88 > 72 \rightarrow$ search continues on β.
- $87 < 88 < 90 \rightarrow$ search continues on β_3.



B+ Trees

Searching for the search key 88.

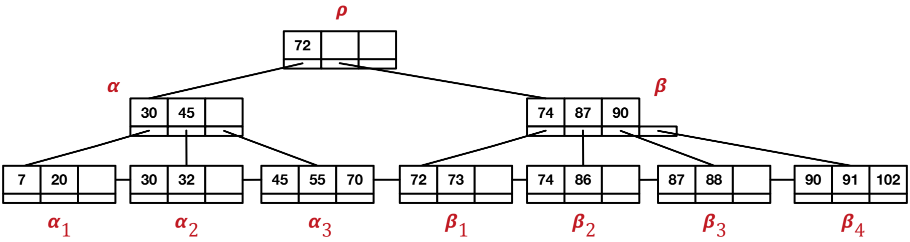
- $88 > 72 \rightarrow$ search continues on β.
- $87 < 88 < 90 \rightarrow$ search continues on β_3.



B+ Trees

Searching for the search key 88.

- $88 > 72 \rightarrow$ search continues on β.
- $87 < 88 < 90 \rightarrow$ search continues on β_3.



B+ Trees – Pros and Cons

Advantages:

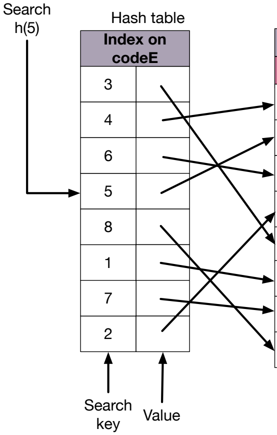
- **Efficient search**: $O(\log n)$
- **Efficient updates**: $O(\log n)$
- **Range queries** are supported.
- Combination of disk and in-memory indexes (levels of the tree).

Disadvantages:

- Even if a search key is found in the internal nodes, the search has to reach the leaves.

Hash Indexes

- An **hash function** is used to map a search key to one **bucket**.
- The hash function is **deterministic**. For the same search key, it produces the same values.



Employee					
codeE	**first**	**last**	**position**	**salary**	**codeD**
4	Mary	Green	Credit analyst	65,000	62
5	William	Russel	Guidance counselour	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
1	Joseph	Bennet	Office assistant	55,000	14
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Assistant director	120,000	14

Hash Indexes – Pros and Cons

Advantages:

- Excellent computational cost: $O(1)$ insertion, deletion, search.

Disadvantages:

- Best when in-memory.
- Impossible to answer range queries.

Using Indexes

- Indexes are key to speed up queries.
- Indexes come with a cost.
  - **Storage space** to store the indexes.
  - **Index updates** when rows are inserted, updated and deleted.
- When you should **avoid using indexes**.
  - **Low-read**, **high write** columns.
  - **Low cardinality** columns (with few distinct values).
  - **Small tables**.

Composite Indexes

Composite indexes can be created in the following cases:

- Columns that **co-occur frequently** in queries.
- High-cardinality columns.

Creating a composite index on (`col1`, `col2`) will **also** create an index on `col1`.

Clustered Vs Non-clustered Indexes

When to create a clustered index.

- Queries SELECT all or most of the columns of the table.
- Frequent JOIN and WHERE on specific columns.
- Columns used frequently in ORDER BY statements.

Remember that:

- A table can only have **at most one** clustered index.
- Usually, the clustered index is the primary index.
- **Avoid** the use of a clustered index for tables that are **frequently updated**.

Create Indexes in SQL

```
CREATE INDEX my_index ON Employee(last)

CREATE UNIQUE INDEX my_index ON Employee(last)

CREATE INDEX my_index ON Employee(last) USING BTREE
```

- On the columns of the primary key a B+ tree index is automatically created.
- Same goes for the UNIQUE constraint.

Transactions

Consistency

- **State** of a database: set of values that are stored in the database.
- **Consistency.** State of the database at a given time when the data meet all the integrity constraints.
  - Salary is not negative.
  - Employees work in an existing department.
  - Same position = same salary.
- **Read** operations do not change the state of the database.
- **Write** operations change the state of the database.
  - Inconsistencies might arise.
- Relational databases approach to consistency: pessimistic.
  - Inconsistencies will happen.
  - The notion of **transaction** is key to maintain consistency.

Transactions

- Suppose we have the following table.

```
BankAccount (account_nbr, balance)
```

- Suppose that the tables contains two rows (1, 350) and (2, 500).
- Assume that we run the two following queries:

```
UPDATE BankAccount SET balance=balance-200 WHERE account_nbr=1;
UPDATE BankAccount SET balance=balance+200 WHERE account_nbr=2;
```

- If another application queries the database between the two queries, it will see an **inconsistent database**.
  - The values of the rows will be (1, 150) and (2, 500), instead of (1, 150) and (2, 700).

Transactions

Definition (Transaction)

A **transaction** is a sequence of read and/or write operations on a database that are executed as a **single atomic operation**. Either all are executed or none. Importantly, the values are stored only if the transaction is successful.

- In our previous example:

Example

```
START TRANSACTION;
UPDATE BankAccount SET balance=balance-200 WHERE account_nbr=1;
UPDATE BankAccount SET balance=balance+200 WHERE account_nbr=2;
COMMIT;
```

Transactions

Transactions have the following properties (ACID):

- **Atomicity (A).** "All or nothing".
- **Consistency (C).** From a consistent state to a consistent state.
  - some operations within the transaction may lead to inconsistencies.
- **Isolation (I).** Serializability of transactions.
- **Durability (D).** Upon commit, all the updates are permanent.

- A relational database enforces **strict consistency** with transactions.
- This might hamper performances in a **distributed database**.