Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Master of Science (MSc)

Applied Information and Data Science

Note: Blue speech bubbles contain additional information for you.

Also, the appendix provides you additional information you might use in your daily job.

An icon of PostgreSQL or MSSQL indicates that the appendix contains syntax differences.

bbv Software Services AG
Georg Lampart
Senior Database Consultant

T direkt +41 41 766 19 71
georg.lampart@hslu.ch

Luzern 15.05.2021

Database Management for Data Scientists
5 – SQL Performance

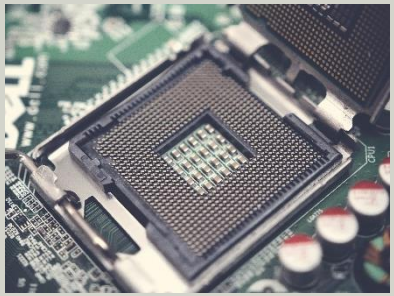
Performance

Time to wait for a query result?

4 PIECES FOR PERFORMANCE

1. I/O vs. Memory
2. Indexes
3. Query Optimizer
4. Good SQL
5. ( )

Hardware components for data – Capacity and Latency (Speed)

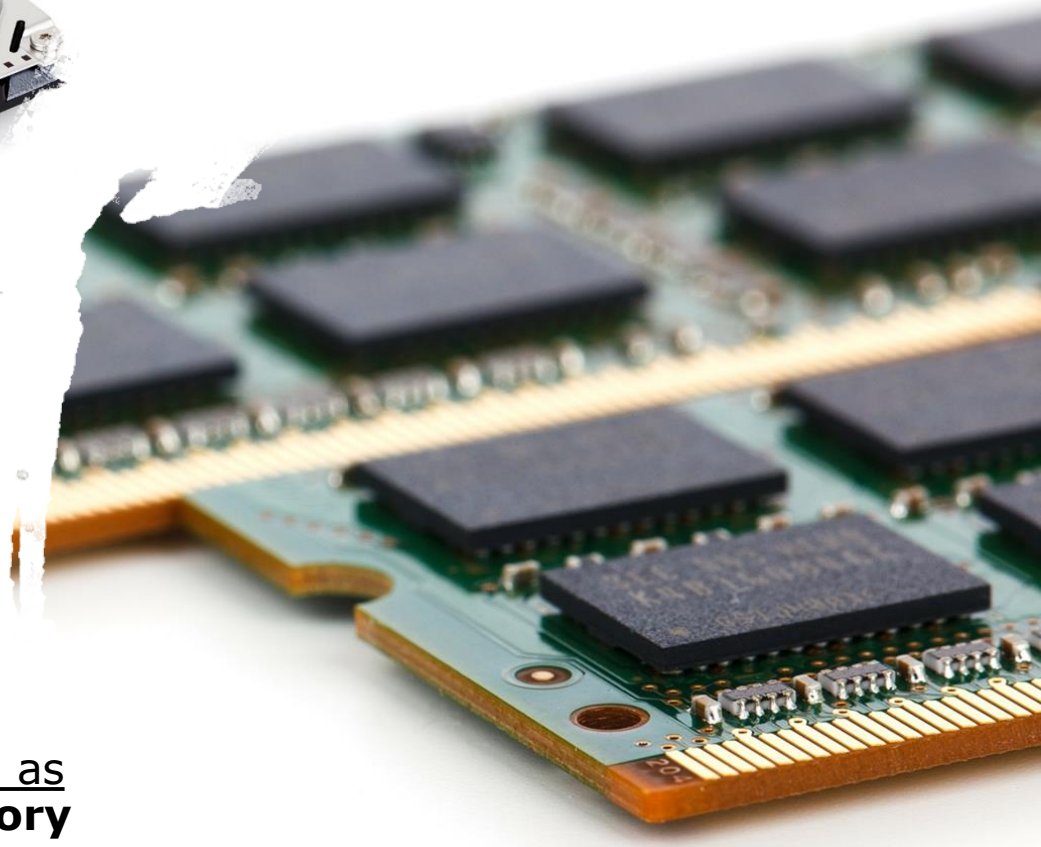
	CPU Cache	RAM/Memory	Disk (HDD/SSD)
			
Capacity	1MB – 50MB	Up to 128/256GB (more is possible)	Up to x TB
Access Latency	1 - 30ns	100 - 1'000ns	1 - 10ms (SSD: 0.05-0.15ms)
Scaled Latency*	**2 sec - 1 min**	**4 - 40 min**	**SSD : 17 hrs - 4 days HDD: 1 - 9 months**

*Baseline:
- 1 CPU Cycle
- Scaled to 1 sec

3-40x slower

1'000x !

https://www.prowesscorp.com/computer-latency-at-a-human-scale/



Consequence

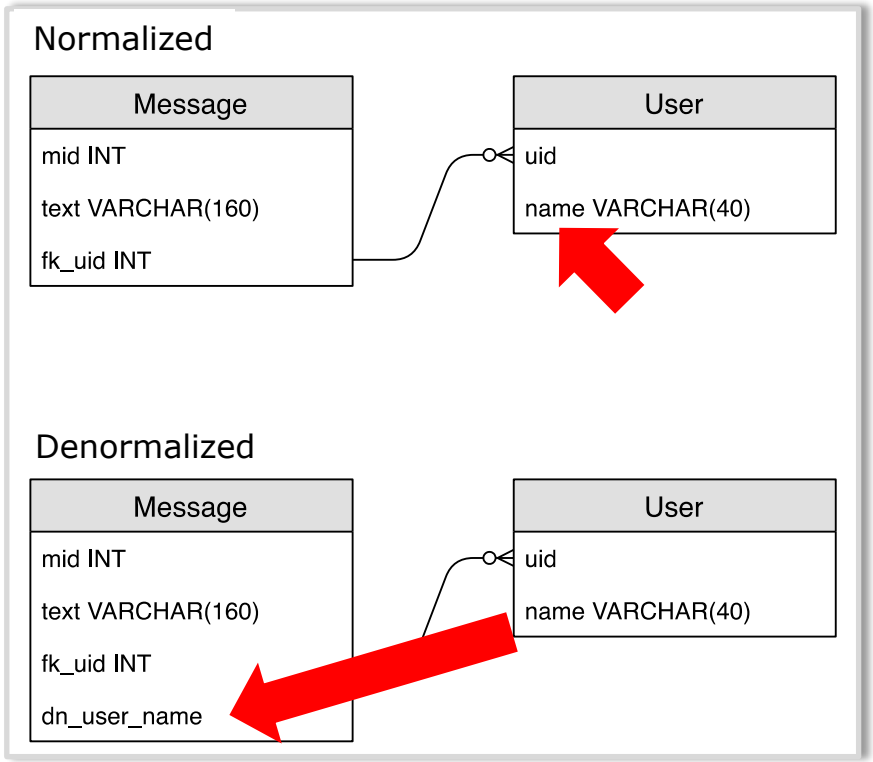
A 300GB database **might <u>not</u> fit** into the fast memory!

Queries should process <u>as **little data** as possible</u> in order to use data in **memory** and **avoid** unnecessary **disk** access!

Table Joins

Table joins can be bottlenecks.
Especially without indexes, they need to process a lot of data and, thus, might heavily use I/O.

https://www.datacamp.com/community/tutorials/sql-tutorial-query



For query performance, you can **skip/avoid** joins by using
- **denormalized** table design
- **materialized view**

https://rubygarage.org/blog/database-denormalization-with-examples

```
SELECT u.name, COUNT(*)
FROM Message AS m
    INNER JOIN User AS u
    ON m.fk_uid = m.uid
GROUP BY u.name;
```

Think web-scale
→ Millions of users
→ Billions of messages

```
SELECT
   m.dn_user_name,
   COUNT(*)
FROM Message AS m
GROUP BY m.dn_user_name;
```

What might be some **drawbacks** if you:

- **denormalize** tables

Locate data quickly / Minimize data access

INDEXES

Question

The **dictionary** is one example of an **index** you might use from time to time.

Do you know **more examples**?

DBMS implement **different types** of indexes. Let's get to know some common index types.

Source: By Langenscheidt - Langenscheidt, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=31355102

Index

Clustered Index

Attributes:

- Tables **physically ordered** by key
- The **key(s)** and the **data** of a row are **stored together**
- **Only one** clustered index per table

Terminology in DBMS:

MySQL:	**Clustered Index**
MSSQL:	Clustered Index
PostgreSQL:	n/a

Metaphor: Dictionary

Source: By Langenscheidt - Langenscheidt, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=31355102

Index

Nonclustered / Secondary Index

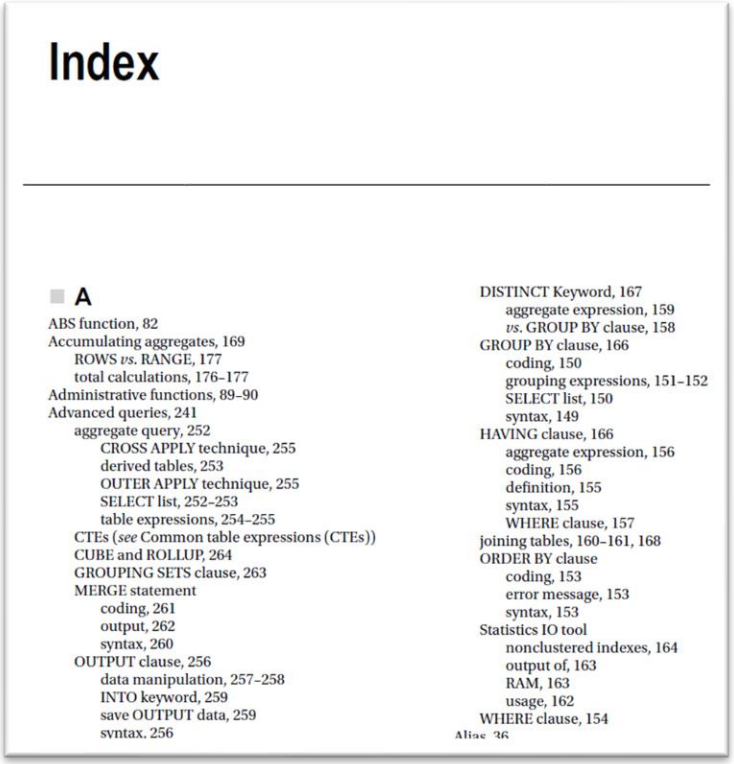
Attributes:

- **Additional** data structure containing key(s)
- **More than one** index possible per table
- Data is **separate** from index

Terminology in DBMS (max. indexes)

MySQL:	**Secondary Index (64)**
MSSQL:	Nonclustered Index (999)
PostgreSQL:	Secondary Index (unlim.)

Metaphor: Keyword index



Index

Heap (not an index)

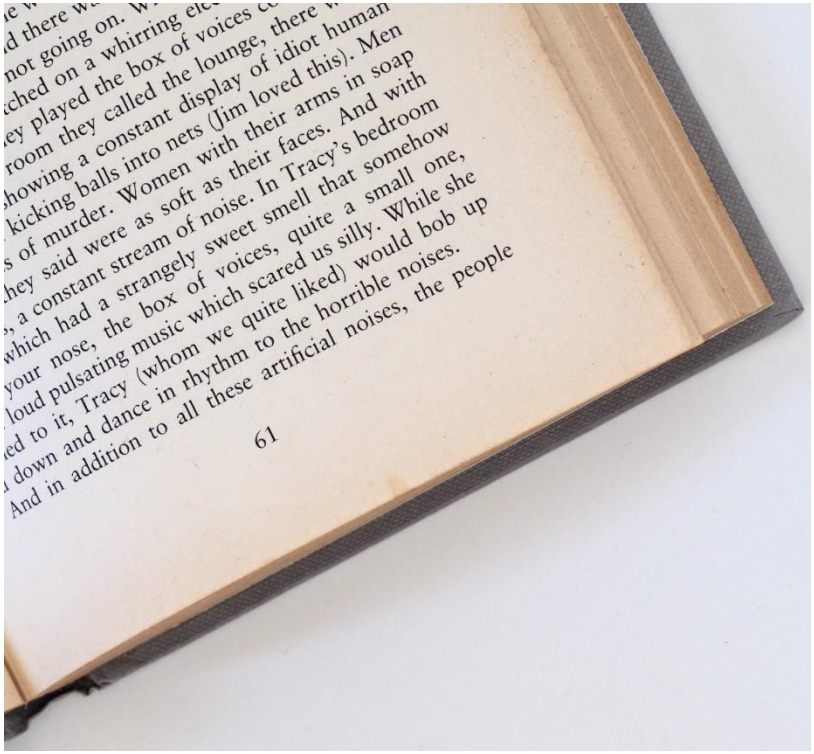
Attributes:

- **Table** content **unordered**
- Data usually **appended** to the end
- **Identifier** necessary to access the row (additional 6 bytes per row!)
  - Row Identifier [RID] or
  - Tuple Identifier [TID]

Terminology in DBMS:

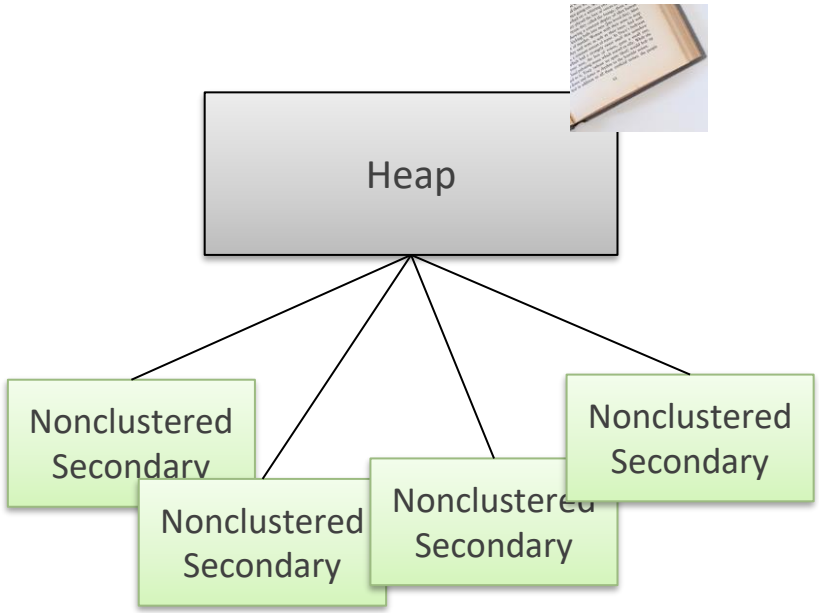
MySQL:	**n/a**
MSSQL:	Heap (Row Identifier)
PostgreSQL:	Heap (Tuple Identifier)

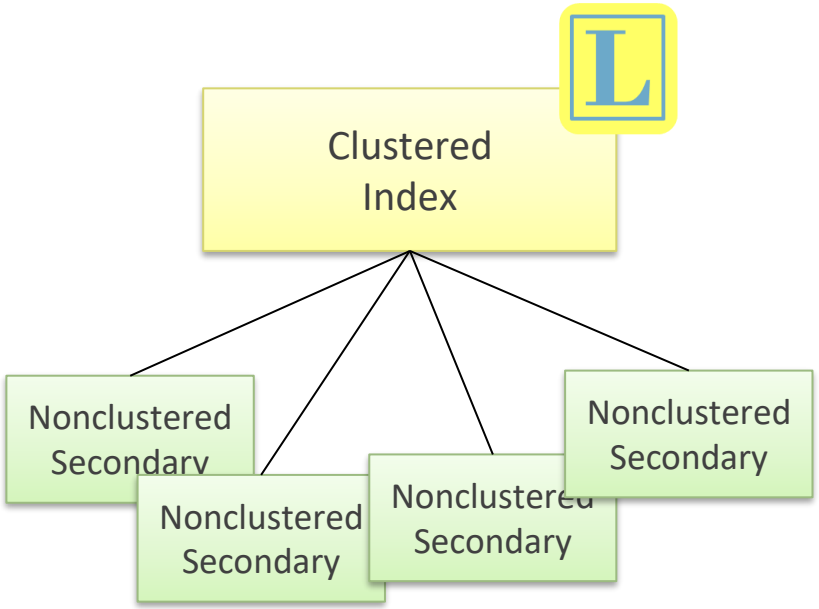
Metaphor: Novel/Notebook



Index

Table and Index layout





PostgreSQL: Only this layout
MSSQL: Possible but not preferred*

MySQL: Only this layout
MSSQL: Preferred*

* MSSQL: As a rule of thumb, start with a clustered index on the primary key. There are only a few scenarios where a heap has advantages (like importing staging data or a real large data warehouse with terabytes of data)

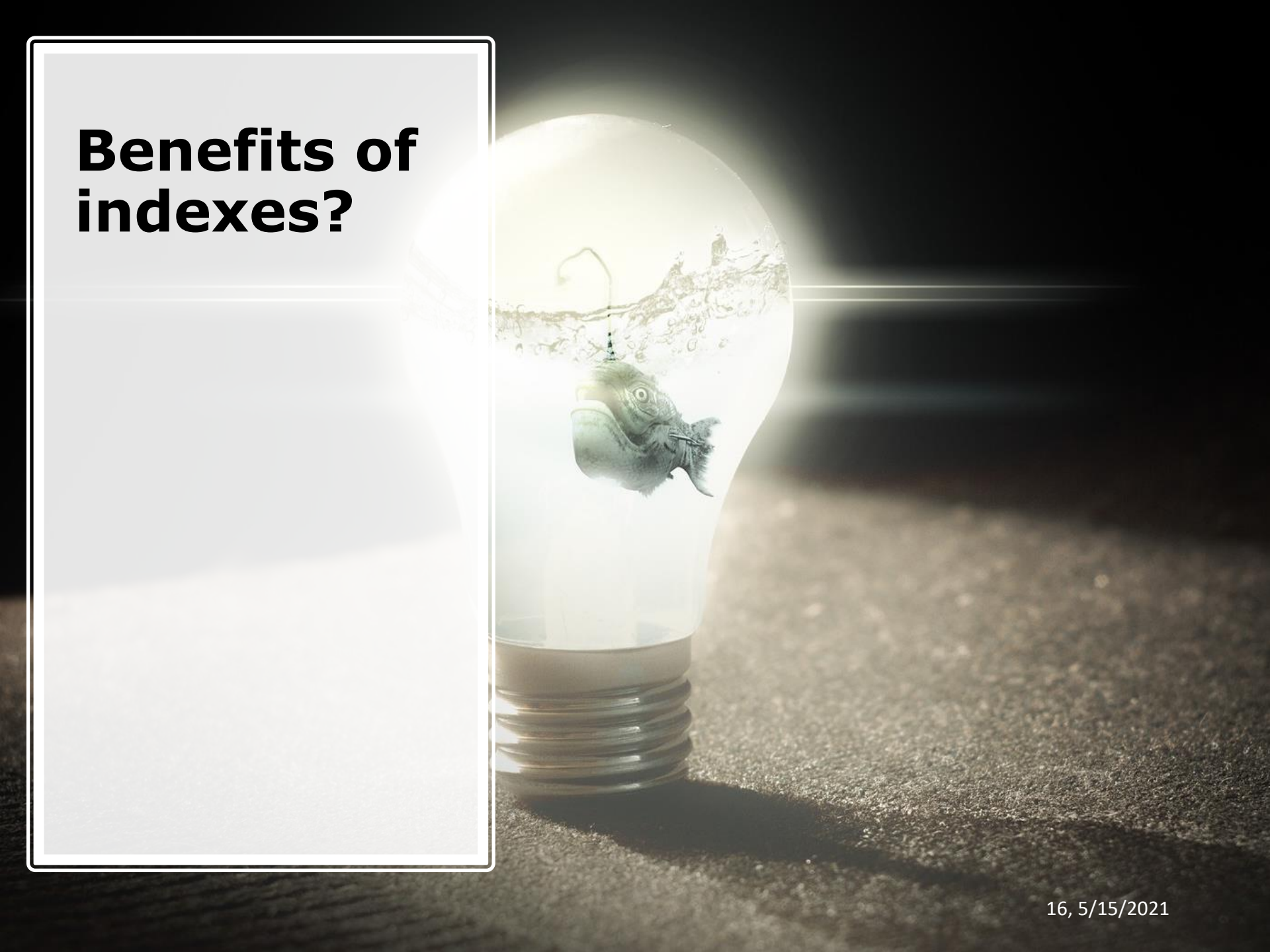
Index

Storage impact

	Heap / Clustered Table	Nonclustered / Secondary Indexes
PostgreSQL SQL Server	Heap 1)	PK → RID 51 → xA 77 → xB 86 → xC 89 → xD 91 → xE Surname → RID Beck → xD Fowler → xB Lerman → xE Martin → xC Randal → xA
MySQL SQL Server	Clustered Index PK / Surname / Firstname 51 Randal Paul S. 77 Fowler Martin 86 Martin Robert C. 89 Beck Kent 91 Lerman Julie	Surname → PK Beck → 89 Fowler → 77 Lerman → 91 Martin → 86 Randal → 51

1) RID (=Row Identifier for MSSQL) or TID(= Tuple Identifier for PostgreSQL)

Benefits of indexes?



Drawbacks of indexes?

Index

SQL – Clustered Index

MySQL:

- MySQL **automatically** clusters the table!
- It uses the **primary key** or – if missing – a **unique** index
- Fallback: it creates a hidden index

```
CREATE TABLE person(
  PersonId INTEGER NOT NULL,
  CONSTRAINT PK_Person
  PRIMARY KEY (PersonId));

ALTER TABLE person
  ADD CONSTRAINT PK_Person
  PRIMARY KEY (PersonId);
```

Note: "PK_Person" is the name (or identifier) of the primary key constraint.

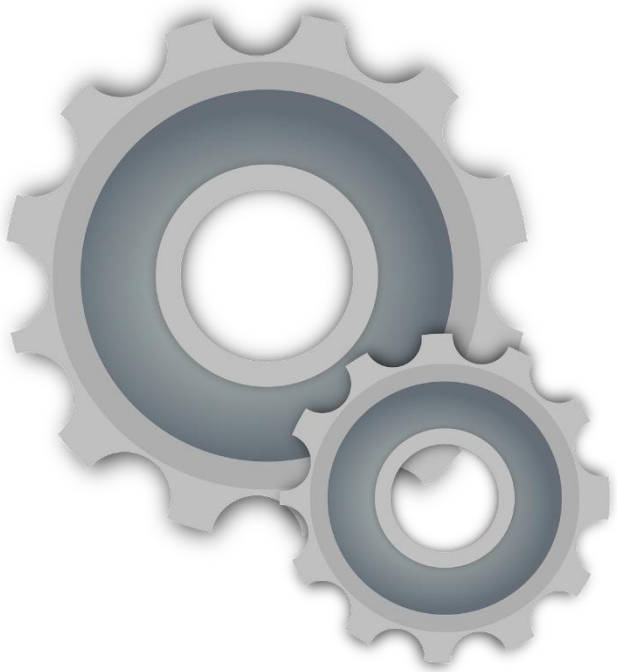
Index

SQL – Nonclustered / Secondary Index

MySQL / PostgreSQL:

- Simple statement!

```
CREATE INDEX
  IDX_Person_Name
  ON person (Name);
```



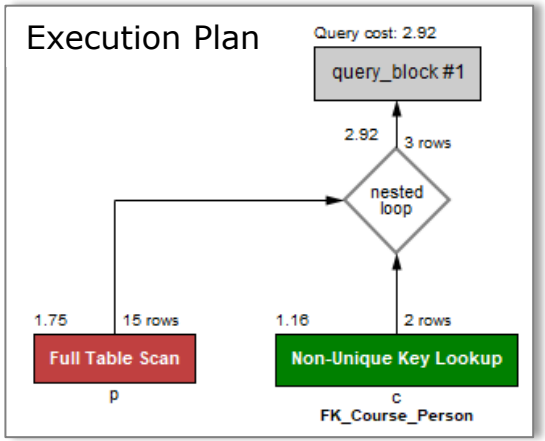
Query Optimizer / Index usage

EXECUTION PLAN

Query Optimizer – Overview

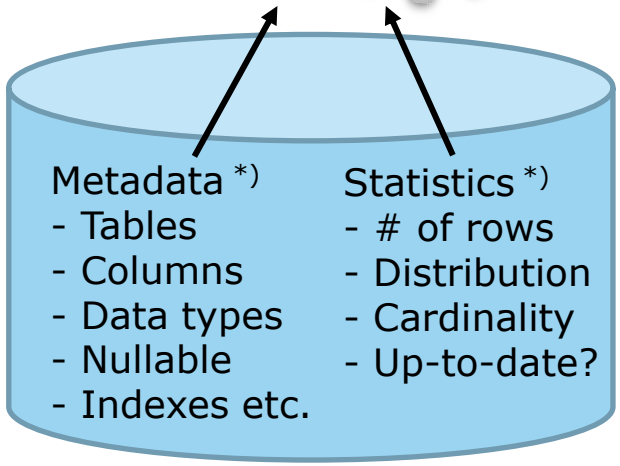
Declarative statement

```
SELECT c.CourseNo, p.Name
FROM course AS c
  INNER JOIN person AS p
  ON c.ProfessorPersonId
   = p.PersonId
WHERE p.Name LIKE 'K%';
```



Result

CourseNo	Name	Birthdate
Die 3 Kritiken	Kant	1950-10-03
Grundzuege	Kant	1950-10-03



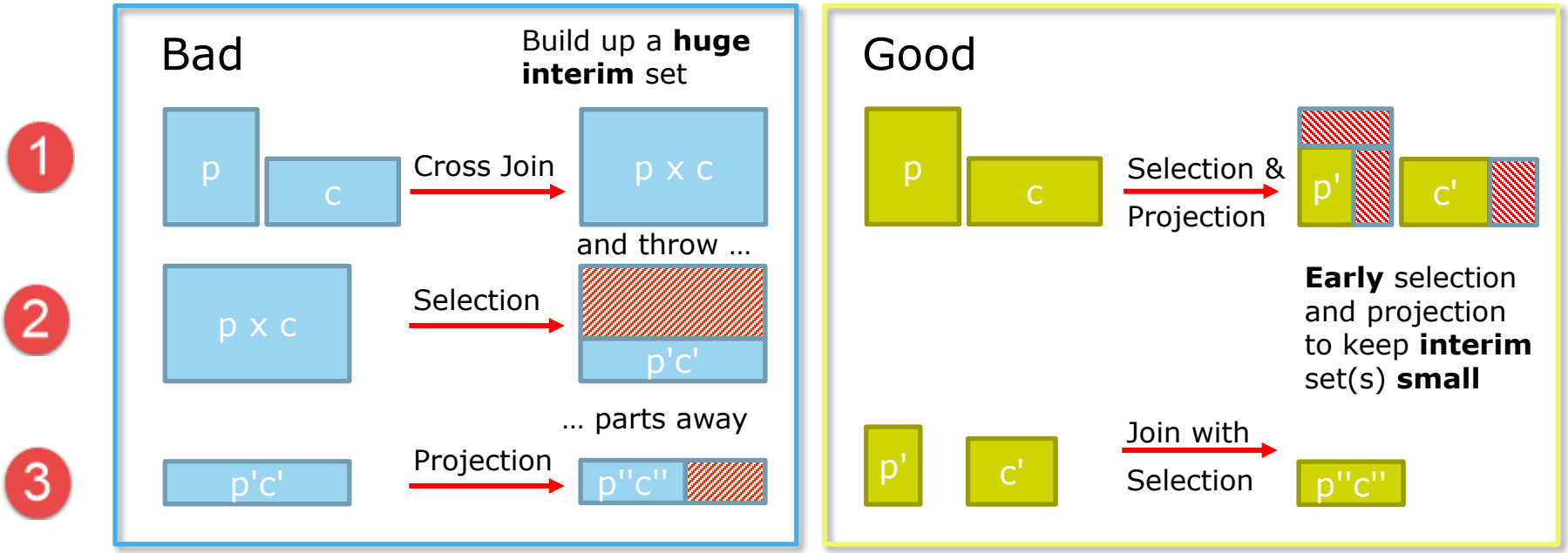
*) We treat this as blackbox. Not part of this course.

MSSQL tip: Your DBA needs to keep the statistics up-to-date

Query Optimizer – Algebraic Transformation

- The Query Optimizer supports you to **reduce the amount** of data processed and also saved in the **interim result** sets
- Consider the following example

```
SELECT c.CourseNo, p.Name, p.Birthdate
FROM course AS c
  INNER JOIN person AS p
  ON c.ProfessorPersonId = p.PersonId
WHERE p.Name LIKE 'K%';
```

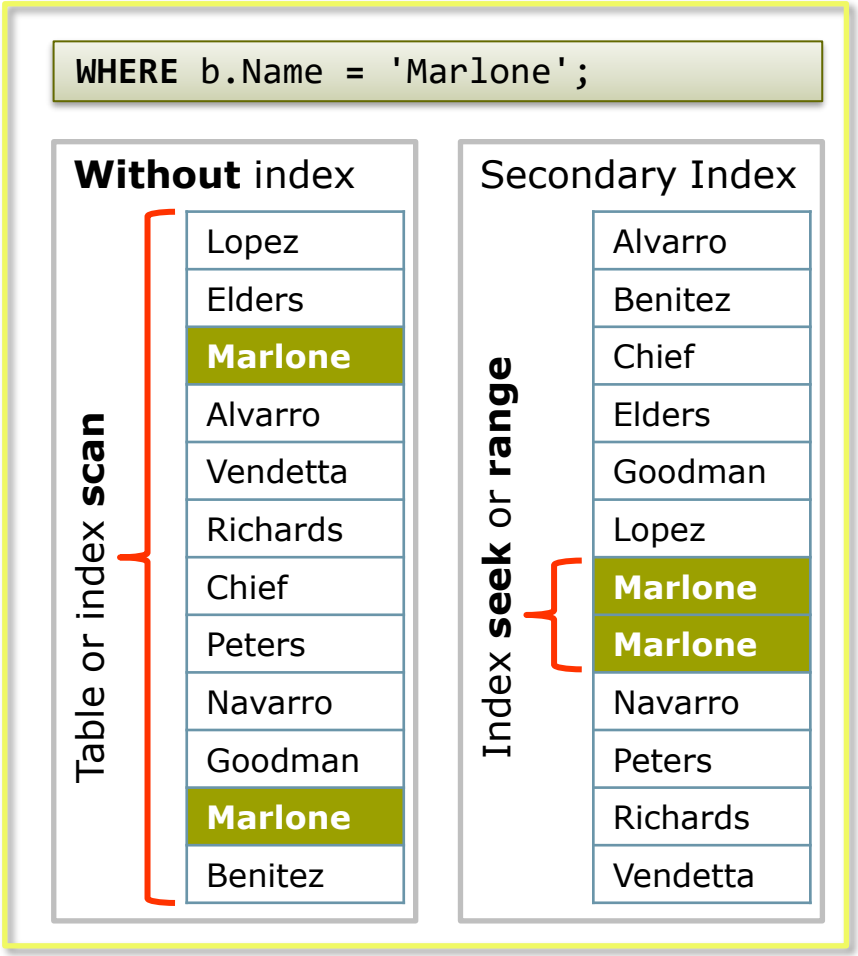


Relational algebra with(out) transformation
see pages 173/174 in Meier/Kaufmann

Query Optimizer normally does this for you!

Query Optimizer – Indexes

- To do its job, the Query Optimizer needs your help
- Build **adequate indexes** that the Optimizer can perform optimal "selection"



In this example, a secondary index on column "Name" helps to find all entries that match "Marlone". The Optimizer only needs to read 2 rows instead of the whole table (= Scan).

With this interim result, the Optimizer can use these 2 rows to join with other tables!

Query Optimizer – Execution Plan

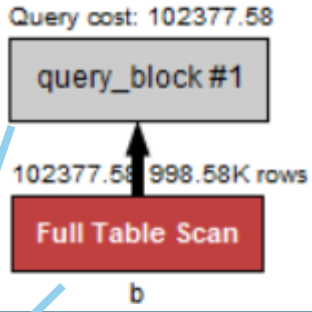
- Most DBMS provide a **textual** or **graphical** execution **plan** **to estimate query performance**

```
SELECT b.PersonId, b.Name, b.Birthdate
FROM bigperson AS b
WHERE b.Name LIKE 'M%';
```

MySQL:

- **EXPLAIN** SELECT …;
- In Workbench: CTRL+ALT+X

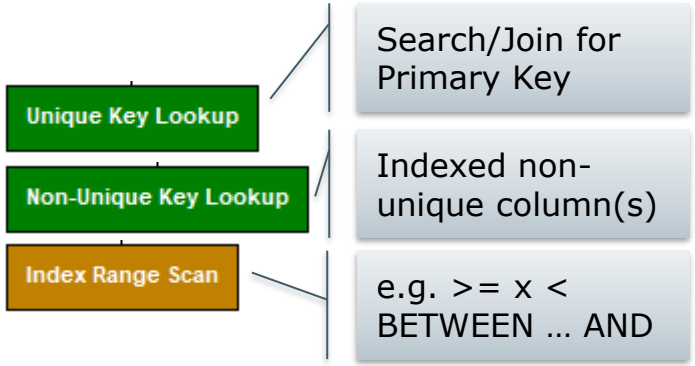
MySQL



Operators

Query Optimizer – Execution Plan Operators

MySQL	Operations/Operators
Good [1] 	Unique Key Lookup Non-Unique Key Lookup Index Range Scan https://dev.mysql.com/doc/workbench/en/wb-performance-explain.html
Bad [1] [2] 	Full Table Scan ("ALL") Full Index Scan ("INDEX")



[1] An operation in the execution plan is often represented by an "operator", e.g. Full Index Scan
[2] On (very) small tables, the Optimizer might choose a Scan because reading the whole table is faster than accessing the index and lookup the data in the table (= 2 seeks per row).
Therefore, on small tables a Scan might not be bad.


Query Optimizer – Some index guidelines

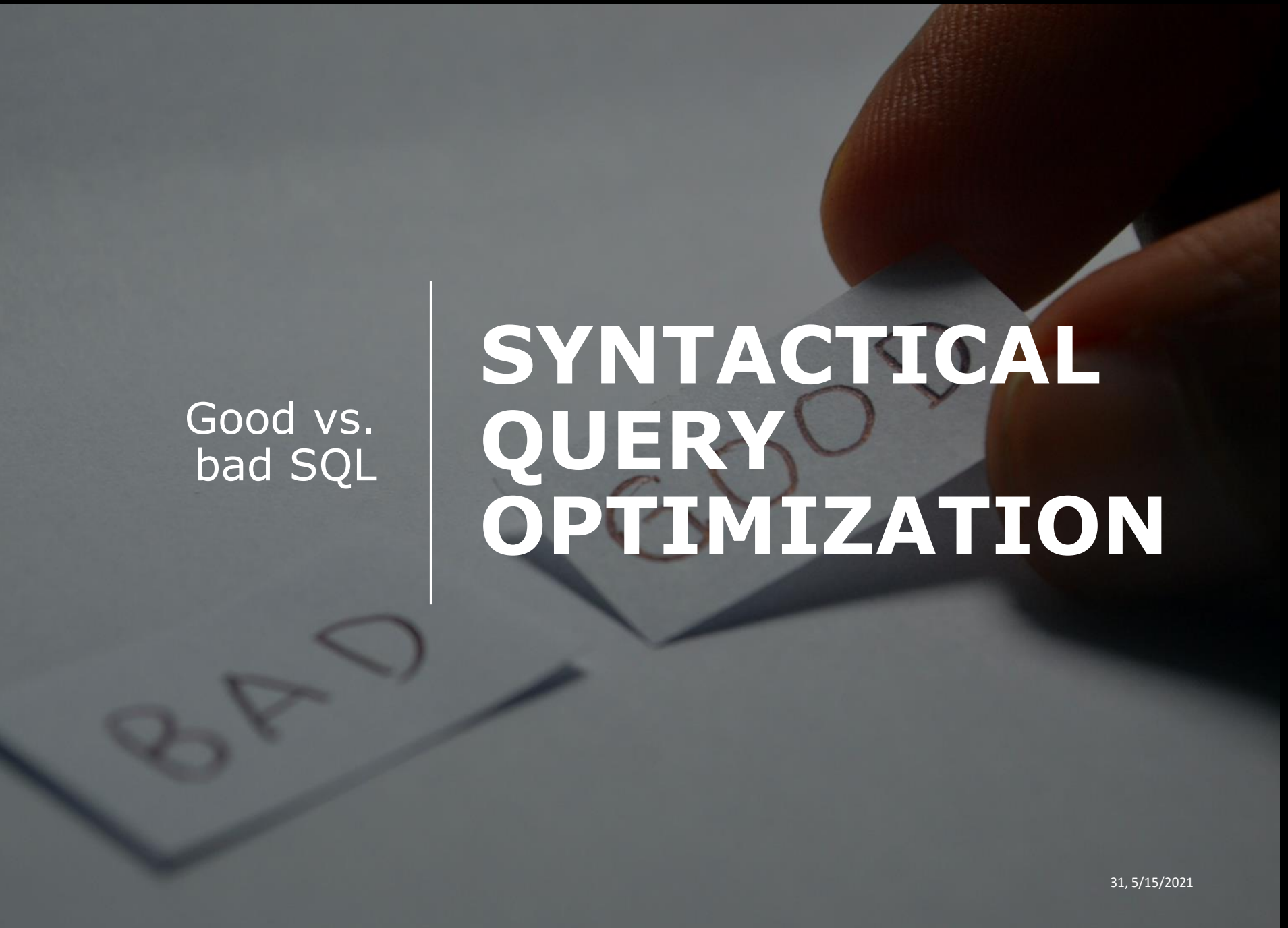
- Generally, only create indexes **that are used** by one or more queries
- **Do not** just **index each** and every **column** in your tables (they must be updated with each insert, update and delete!). Create indexes that **serve several different** queries!
- Use the Execution **plan** to **check** whether the **index is used** for your SQL statement (test with enough data)
- When you **load** a huge amount of data, you might **drop** the indexes and **recreate** them after the import

MySQL

MySQL:

- Note that MySQL **automatically** creates indexes on **foreign key columns** (unless the column already has an index)

These are guidelines. In your project, there might be other rules. Make sure to know the differences and do not just blindly apply new guidelines.



Good vs. bad SQL

SYNTACTICAL QUERY OPTIMIZATION

Question

The following statement is **not** Optimizer-friendly. How could you **improve** it?

```
SELECT COUNT(*)
FROM bigperson AS b
WHERE LEFT(b.Name, 2) = 'An';
```

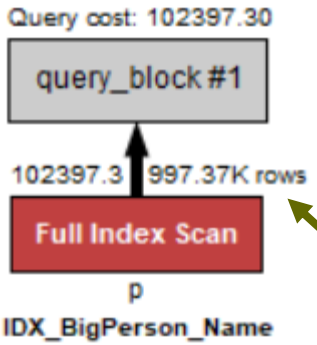
The Optimizer does the best job possible. Assist the Optimizer with optimal SQL code!

Good vs. Bad

Do <u>not</u> use functions on search columns (1)

```
SELECT COUNT(*)
FROM bigperson AS b
WHERE LEFT(b.Name, 2) = 'An';
```

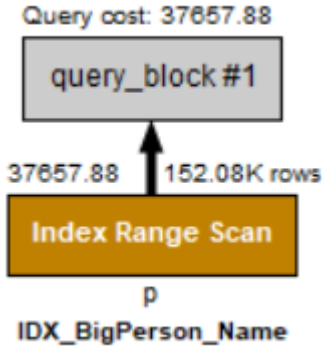
The Optimizer actually uses the index. But it needs to **scan** the index and **executes the function** on <u>each</u> value.



```
SELECT COUNT(*)
FROM bigperson AS b
WHERE b.Name LIKE 'An%';
```

The Optimizer can read the range
>= "An"
< "AO"

Estimated number of rows (997K vs. 152K) !



Good vs. Bad

SARGable queries

- Pseudo acronym = search argument can take **advantage** of an index
- SARGable queries **speed up** execution time, **reduce resource** usage and **minimize blocking** other users
- Consider writing SARGable queries
- A lot of non-SARGable queries **can be rewritten** into SARGable queries

Good vs. Bad

```
SELECT COUNT(*)
FROM bigperson AS b
WHERE YEAR(b.Birthdate) = 1960;
```

```
SELECT COUNT(*)
FROM bigperson AS b
```

Good vs. Bad

Do not use functions on search columns (2)

```
SELECT COUNT(*)
FROM bigperson AS b
WHERE YEAR(b.Birthdate) = 1960;
```

```
SELECT COUNT(*)
FROM bigperson AS b
WHERE b.Birthdate >= '1960-01-01'
  AND b.Birthdate < '1961-01-01';
```

Note: Do not use BETWEEN … AND for data comparison because it is translated into >= …<=.

Example business requirement: Get data of the first two months in 2020 (spot the bug):

x BETWEEN '2020-01-01' AND '2020-**02-28**'

vs.

x >= '2020-01-01' AND x < '2020-03-01'

Good vs. Bad

Do not use functions on search columns (2)

```
SELECT COUNT(*)
FROM bigperson AS b
WHERE YEAR(b.Birthdate) = 1960;
```

```
SELECT COUNT(*)
FROM bigexamresult AS e
WHERE e.ExamNote + @Adjust = 4.0;
```

```
SELECT COUNT(*)
FROM bigperson AS b
WHERE b.Birthdate >= '1960-01-01'
  AND b.Birthdate < '1961-01-01';
```

```
SELECT COUNT(*)
FROM bigexamresult AS e
```

Good vs. Bad

Do not use functions on search columns (2)

```
SELECT COUNT(*)
FROM bigperson AS b
WHERE YEAR(b.Birthdate) = 1960;
```

```
SELECT COUNT(*)
FROM bigexamresult AS e
WHERE e.ExamNote + @Adjust = 4.0;
```

```
SELECT COUNT(*)
FROM bigperson AS b
WHERE b.Birthdate >= '1960-01-01'
  AND b.Birthdate < '1961-01-01';
```

```
SELECT COUNT(*)
FROM bigexamresult AS e
WHERE e.ExamNote = 4.0 - @Adjust;
```

No calculation on search columns!

Good vs. Bad

```
SELECT COUNT(*)
FROM bigperson AS b
WHERE b.ExternalCRM_No = 41;
```

BigPerson

	Column Name	Data Type	Allow Nulls
🔑	PersonId	int	☐
	Name	nvarchar(100)	☐
	Birthdate	date	☑
	ExternalCRM_No	varchar(20)	☑
			☐

```
SELECT COUNT(*)
FROM bigperson AS b
```

Good vs. Bad

Avoid implicit type conversion

```
SELECT COUNT(*)
FROM bigperson AS b
WHERE b.ExternalCRM_No = 41;
```

BigPerson

	Column Name	Data Type	Allow Nulls
🔑	PersonId	int	☐
	Name	nvarchar(100)	☐
	Birthdate	date	☑
	ExternalCRM_No	varchar(20)	☑
			☐

```
SELECT COUNT(*)
FROM bigperson AS b
WHERE b.ExternalCRM_No = '41';
```

Minimize the amount of data to process with **good SQL** and **indexes** (Lamborghini). Otherwise, you get slow queries due to huge amount of data (Excavator)

Processed data matters

4 PIECES FOR PERFORMANCE

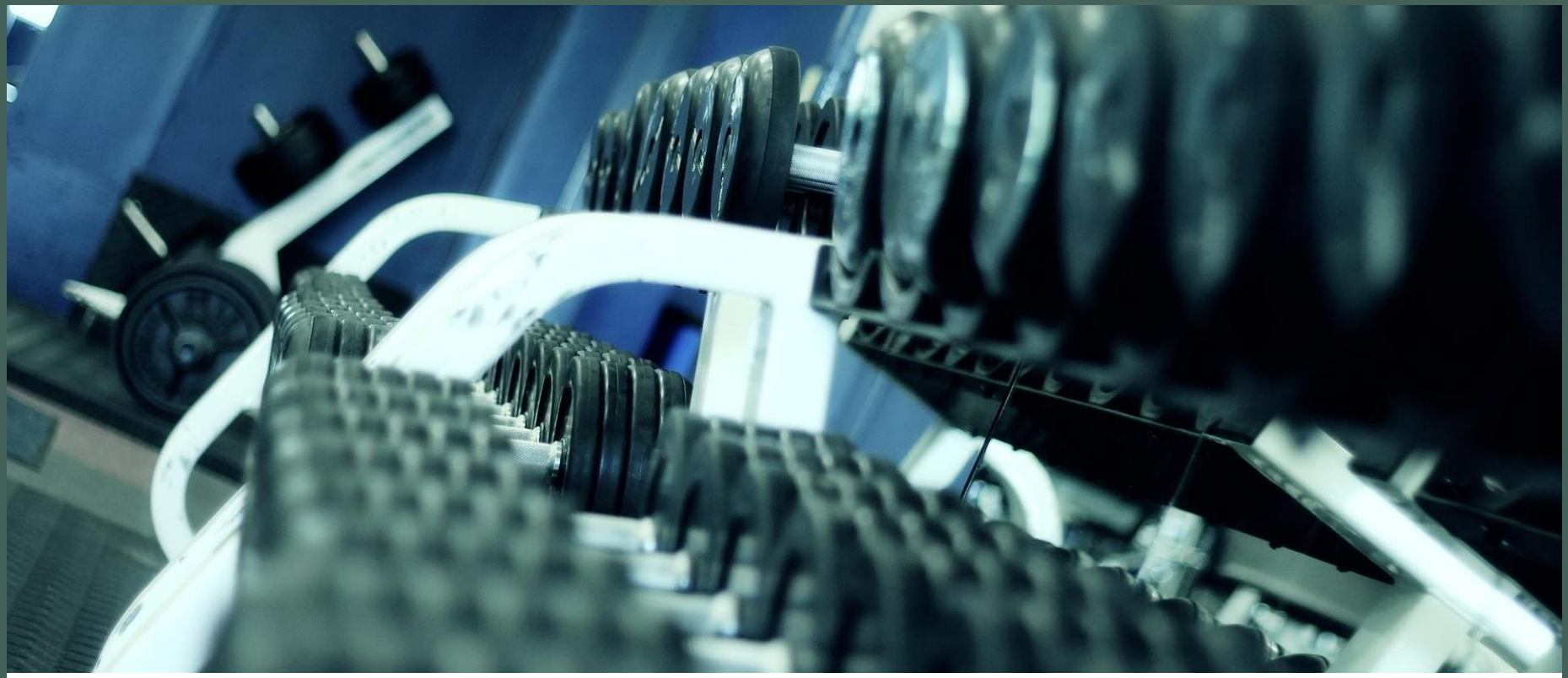
1. I/O vs. Memory
2. Indexes
3. Query Optimizer
4. Good SQL
5. *(Table design/ERD)*

Learning objectives 5: SQL Performance

1. Understand **bottlenecks** (I/O vs. memory, amount of data, joins)
2. Apply **indexes** (types, layout and access pattern) to optimize query performance
3. Show **execution plans** and recognize **good/bad operators** for optimizing queries (explain, scan vs. seek)
4. **Syntactical** SQL performance optimization (good vs. bad SQL, **sargable** queries)

Q & A

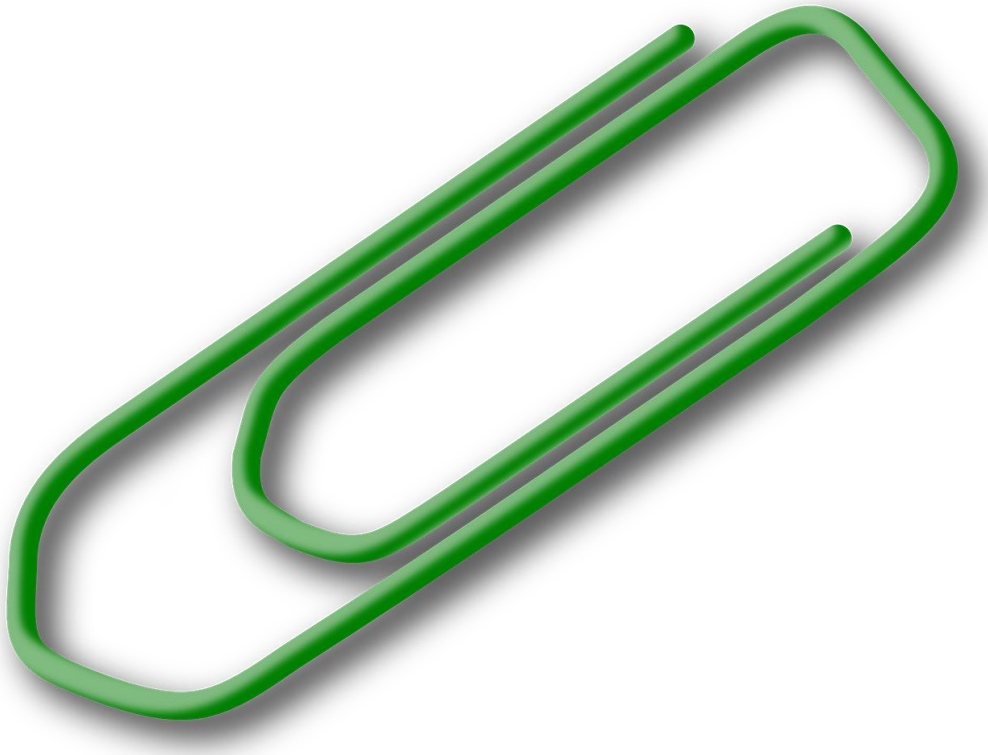
Thank you!



EXERCISE

APPENDIX

MSSQL / PostgreSQL



SQL – Clustered Index (MSSQL syntax)

MySQL:

- MySQL **automatically** clusters the table!
- It uses the **primary key** or – if missing – a **unique** index
- Fallback: it creates a hidden index

```
CREATE TABLE person(
  PersonId INTEGER NOT NULL,
  CONSTRAINT PK_Person
  PRIMARY KEY (PersonId));

ALTER TABLE person
  ADD CONSTRAINT PK_Person
  PRIMARY KEY (PersonId);
```

MSSQL:

- **Recommended** to **explicitly** define the clustered index
- As a **starting point**, use the **primary key** as clustered index
- But you can define **any** column(s) as clustered index (MSSQL will make it unique behind the scenes)

```
CREATE TABLE Course(
  CourseId INTEGER NOT NULL,
  CourseNo VARCHAR(20) NOT NULL,
  CONSTRAINT PK_Course
  PRIMARY KEY CLUSTERED (CourseId));

ALTER TABLE Course
  ADD CONSTRAINT PK_Course
  PRIMARY KEY CLUSTERED (CourseId);
```

```
CREATE CLUSTERED INDEX
IDX_CourseNo ON Course(CourseNo);
```

SQL – Nonclustered / Secondary Index (MSSQL syntax)

MySQL / PostgreSQL:

- Simple statement!

```
CREATE INDEX
  IDX_Person_Name
  ON person (Name);
```

MSSQL:

- **Recommended** to **explicitly** use the keyword NONCLUSTERED!

```
CREATE NONCLUSTERED INDEX
  IDX_Person_Name
  ON Person (Name);
```

Query Optimizer – Execution Plan (MSSQL / PostgreSQL)

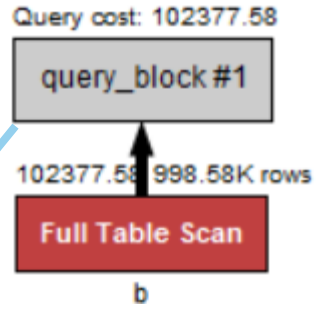
- Most DBMS provide a **textual** or **graphical** execution **plan** **to estimate query performance**

```
SELECT b.PersonId, b.Name, b.Birthdate
FROM bigperson AS b
WHERE b.Name LIKE 'M%';
```

MySQL:

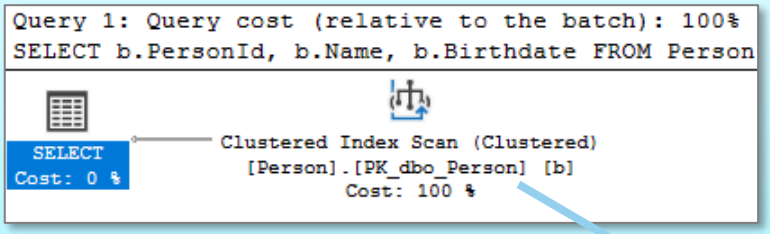
- **EXPLAIN** SELECT …;
- In Workbench: CTRL+ALT+X

MySQL



MSSQL:

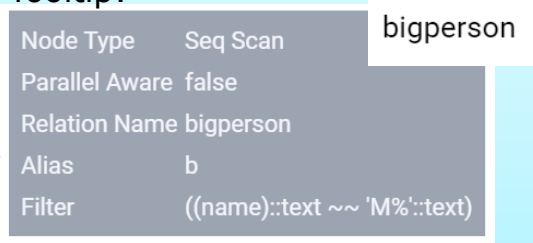
- SET XML STATISTICS ON;
- In SSMS: **CTRL+M**



PostgreSQL:

- **EXPLAIN [ANALYZE]** SELECT …;
- In pgadmin: F7

Tooltip:



Operators

Query Optimizer – Execution Plan Operators

DBMS	Bad operators [1) 2)]	Good operators
MSSQL	• Clustered Index scan • Index scan / Table scan • Key / RID Lookup	• Index seek • Clustered Index seek
MySQL	• Full Table Scan ("ALL") • Full Index Scan ("INDEX") Bad operators are red-colored	• Unique Key Lookup • Non-Unique Key Lookup • Index Range Scan https://dev.mysql.com/doc/workbench/en/wb-performance-explain.html
PostgreSQL	• Seq Scan (= Table scan) • Index Scan • Index Only Scan You need to dig deeper whether an "Index (Only) Scan" is good or bad. Use Explain Analyze to output the rows affected and the costs.	

1) A step in the execution plan is called "operator", e.g. Full Index Scan
2) On (very) small tables, the Optimizer might choose a Scan because reading the whole table is faster than accessing the index and lookup the data in the table (= 2 seeks per row)

Query Optimizer – Some index guidelines (MSSQL / PostgreSQL)

- Generally, only create indexes **that are used** by one or more queries
- **Do not** just **index each** and every **column** in your tables (they must be updated with each insert, update and delete!)
- Use the Execution **plan** to **check** whether the **index is used** for your SQL statement (test with enough data)

These are guidelines. In your project, there might be other rules. Make sure to know the differences and do not just blindly apply new guidelines.

MySQL

MySQL:

- Note that MySQL **automatically** creates indexes on **foreign key columns** (unless the column already has an index)

MSSQL – Recommendations:

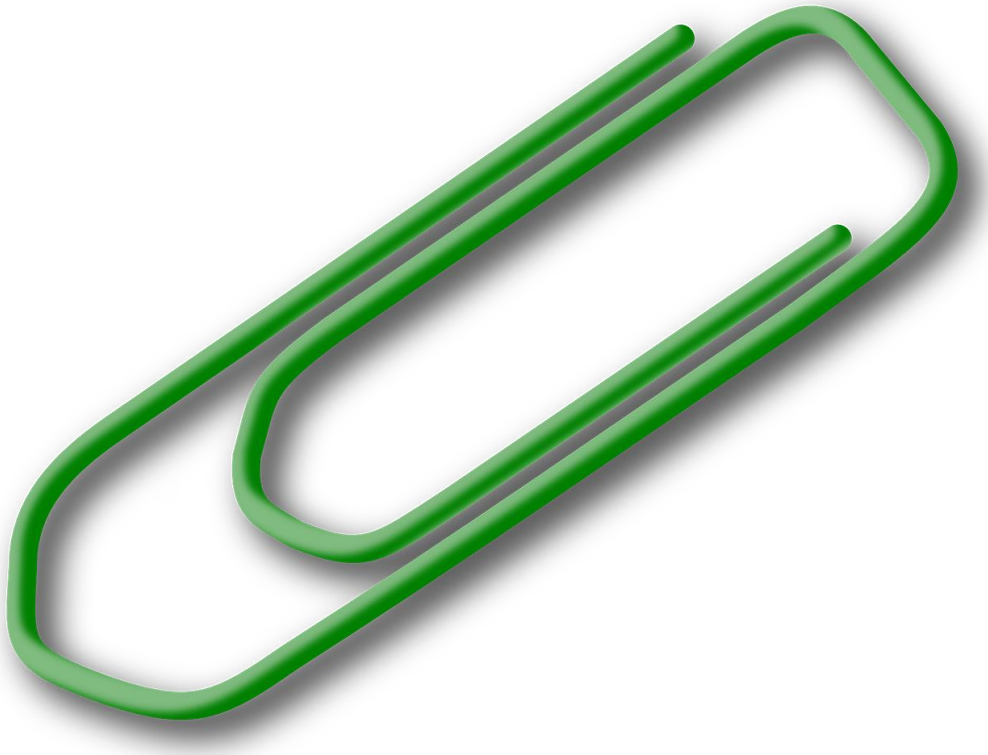
- Create indexes on **foreign key columns** (MSSQL will not do it automatically)
- Create **clustered indexes** on your tables (normally on the primary key)

PostgreSQL - Recommendations:

- Again, create indexes on **foreign key** columns

APPENDIX

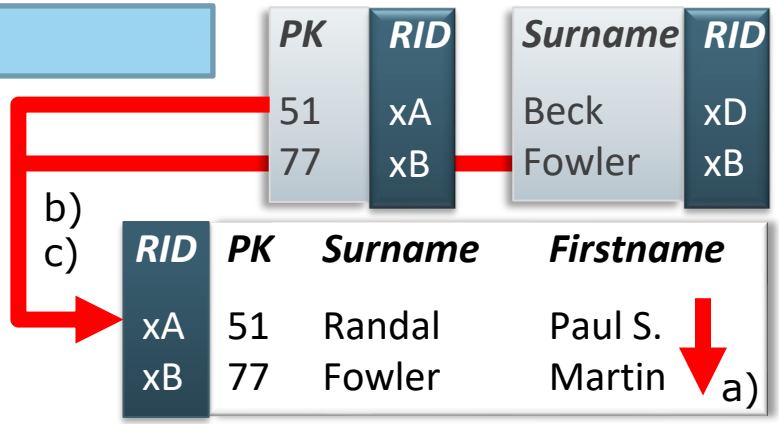
Additional information



Index

Access data – Number of seeks (vs. a scan)

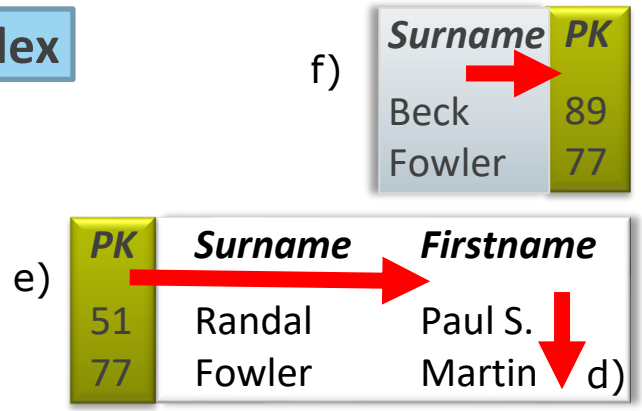
Heap



Search process	Access
a) Find Firstname	Heap Scan
b) Find Firstname by PK	2 Seeks
c) Find PK by Surname	2 Seeks

b) c): Find RID (or TID) in the secondary index; then seek the row in the heap to read the requested data (Firstname or PK)

Clustered Index



Search process	Access
d) Find Firstname	Table Scan
e) Find Firstname by PK	1 Seek
f) Find PK by Surname	1 Seek

a) d): Column "Firstname" has no secondary index. The table (heap or clustered) is always scanned.