Digital Career Institute

Databases - Basic Performance

Digital Career Institute

Database Performance

Database Performance

How good a database performs depends on two main factors:

Throughput	Workload
Offer	Demand

The **throughput** is the amount of operations per unit of time that the system can *offer*.

The **workload** is the amount of operations per unit of time that the applications may *demand*.

Database Throughput

Database Throughput

It is the overall capability of the hardware and software to process data.

Hardware

Software

A better hardware will often translate into a better throughput.

The throughput can also be optimized using software strategies.

Hardware Optimization

DLI

The most relevant hardware parts for the database are:

Hard Drive

Memory

Data is stored on the file system in the hard drive. A faster hard drive will produce faster results.

Some data is often also stored in memory. A faster memory module will produce faster results.

Hardware Optimization

Multiple hardware can be used to increase the throughput.

Clusters

A database cluster is a set of database instances that work together to provide a single point of entry solution.

Database clusters usually act as routers to deliver each request to the instance with the lowest demand at each moment.
This is called **load balancing**.

Software Optimization

Optimizations may be made on two main areas:

Accessing Data

Searching Data

Accessing the data takes time, more so if the data is in the file system. RDBMS store some data in memory to improve the throughput.

If the data is stored with a more specific structure, searching may require less operations and less reading may be required.

Accessing Data: Hard Drive vs. RAM

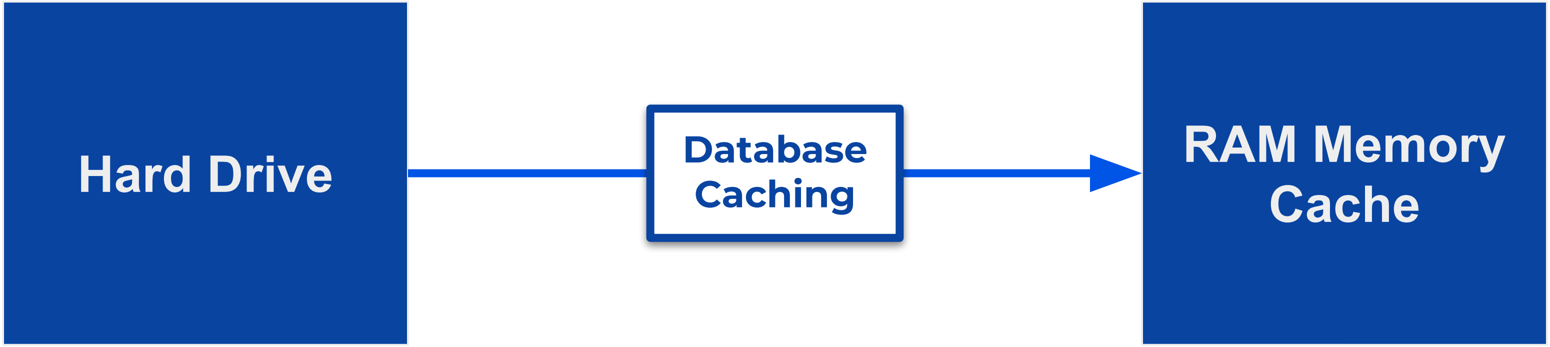
Accessing the hard drive consumes more time and resources than accessing the memory.

	Hard Drive	RAM
Speed	**750MB/sec.**	**12.800MB/sec.**

The durable ACID property requires the data to be stored on the hard drive. Every time we read, update or create contents on the database, the hard drive needs to be accessed. But some operations can be used only in-memory and some data can be stored in memory.

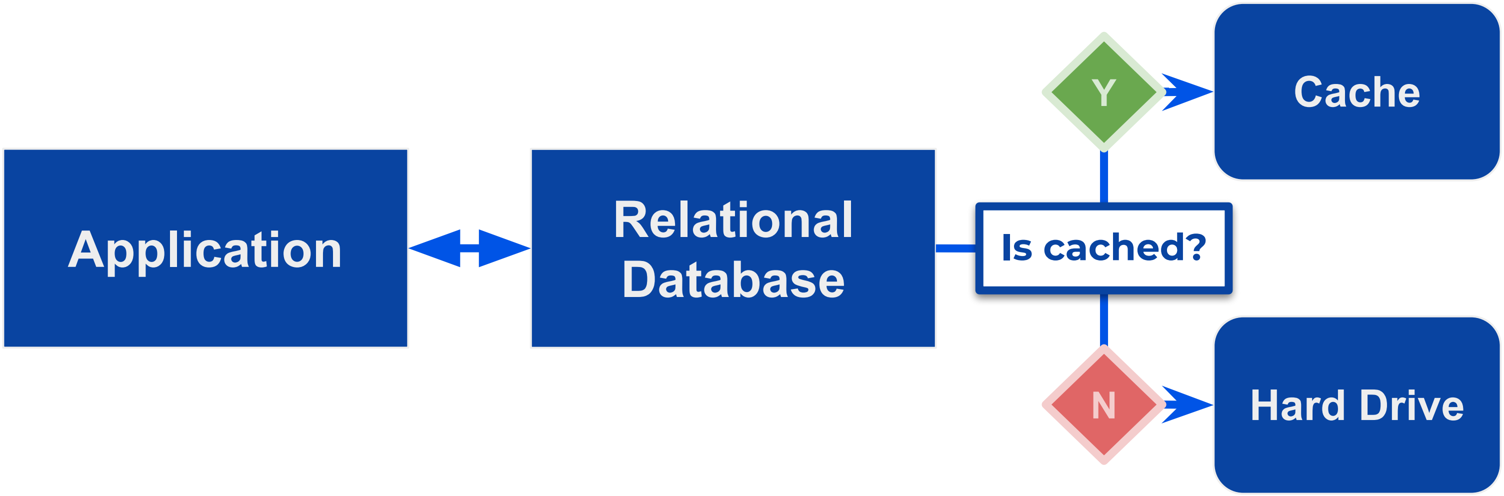
Accessing Data: Caching

The process of storing a copy of the most common data to improve access times is called **caching**.



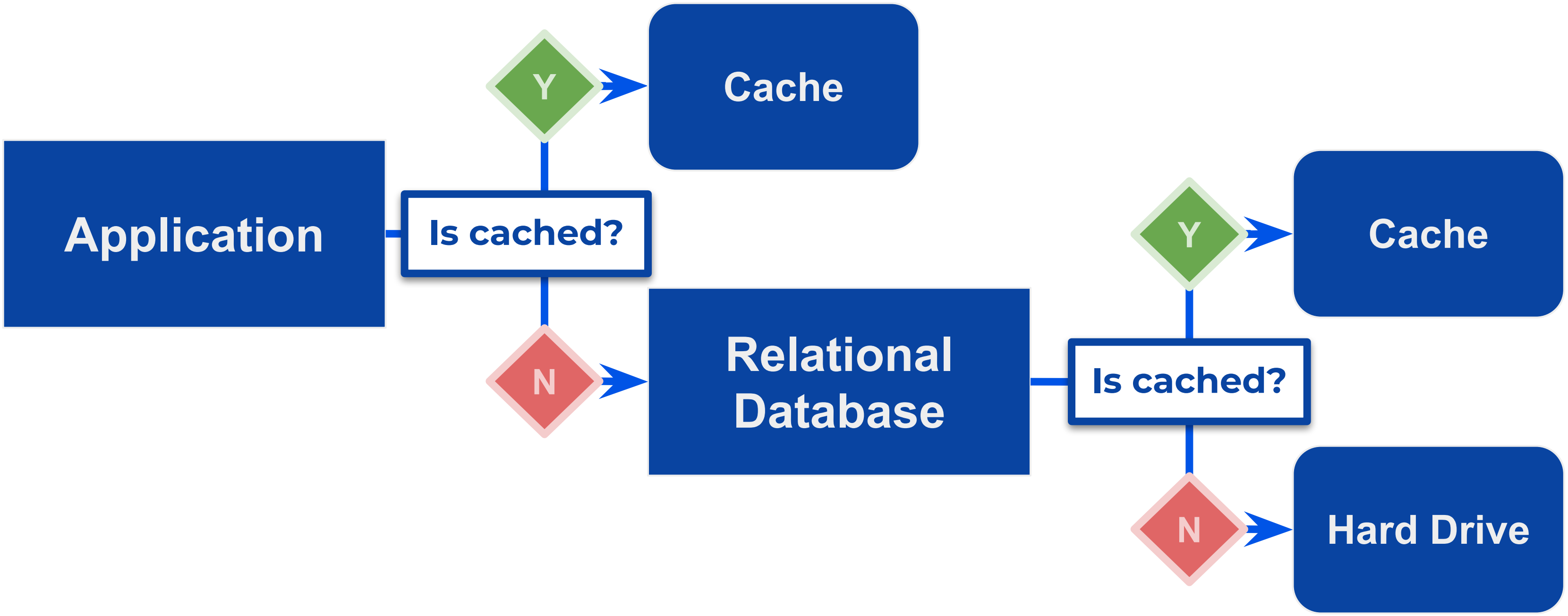
Accessing Data: Caching

If the required content has been cached, the RDBMS will access the in-memory copy. If not, it will access the hard disk.



Accessing Data: Caching

Caching may happen at different levels of the software stack.



Searching Data

Test 1.

Count the number of passes in the white team.

In the next slide, you will see a video of a white team and a black team of 3 players each, all mixed up and moving around.

Each team has a ball and passes the ball only between them.

Searching Data

Searching Data

Had you seen this picture before the video, you would probably have been able to notice the gorilla and, at the same time, count the correct number of passes.

Because nobody told you there would be a gorilla, you may have missed it. But you probably counted the passes right, because you were warned in advance.

An RBDMS also performs better when it knows what it will be expected to do.

Searching Data

Test 2.

Count the number of gorillas in the list.

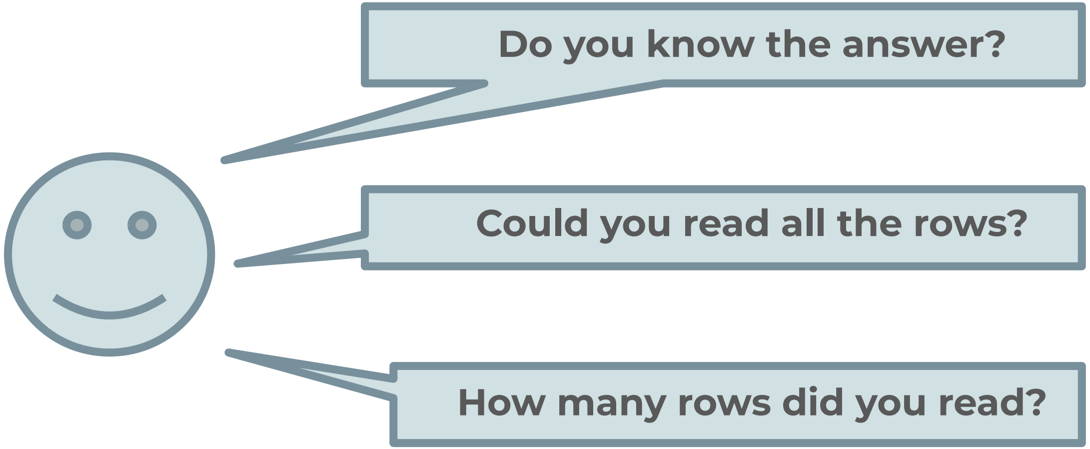
On the next slide, you will see a list of 17 players
of three teams: White, Black and Gorilla.

The slide should only be visible for 2 seconds.

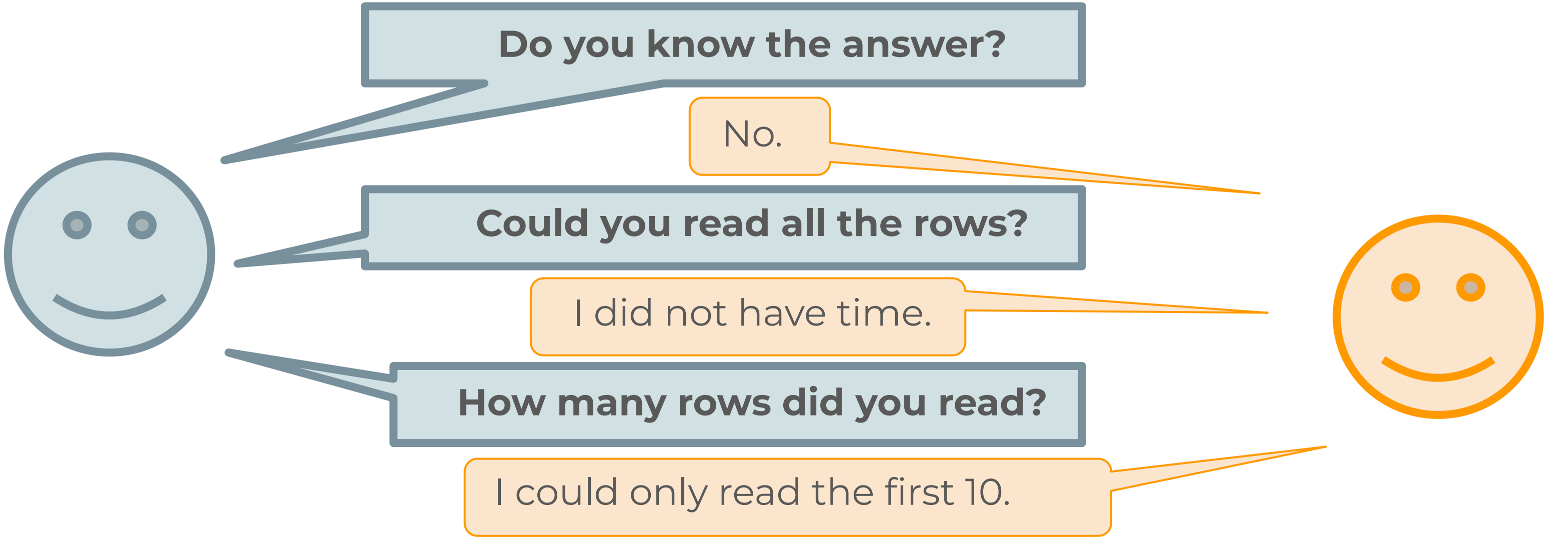
Searching Data

Name	Team
Allison	White
Alicia	Gorilla
Daniela	White
Charles	Gorilla
Johan	Black
Patrick	White
Julia	Gorilla
Carmen	Black
Ellen	White
George	Gorilla
Gemma	Black
Elliot	White
Sean	White
Samantha	Black
Michael	White
Jennifer	Black
Martha	White

Searching Data



Searching Data



Searching Data

Test 3.

Count the number of gorillas in the list.

You will see the same list.

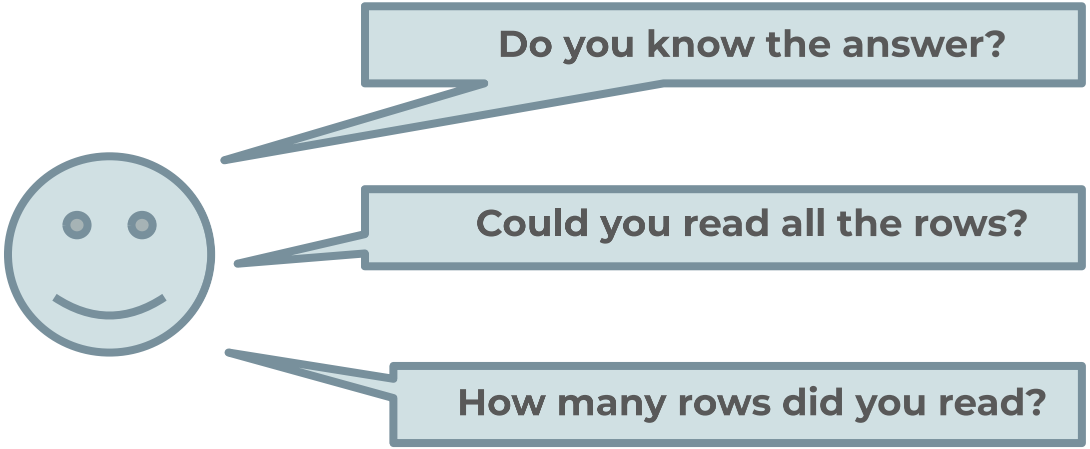
This time, the list will be sorted by team name.

The slide should only be visible for 2 seconds.

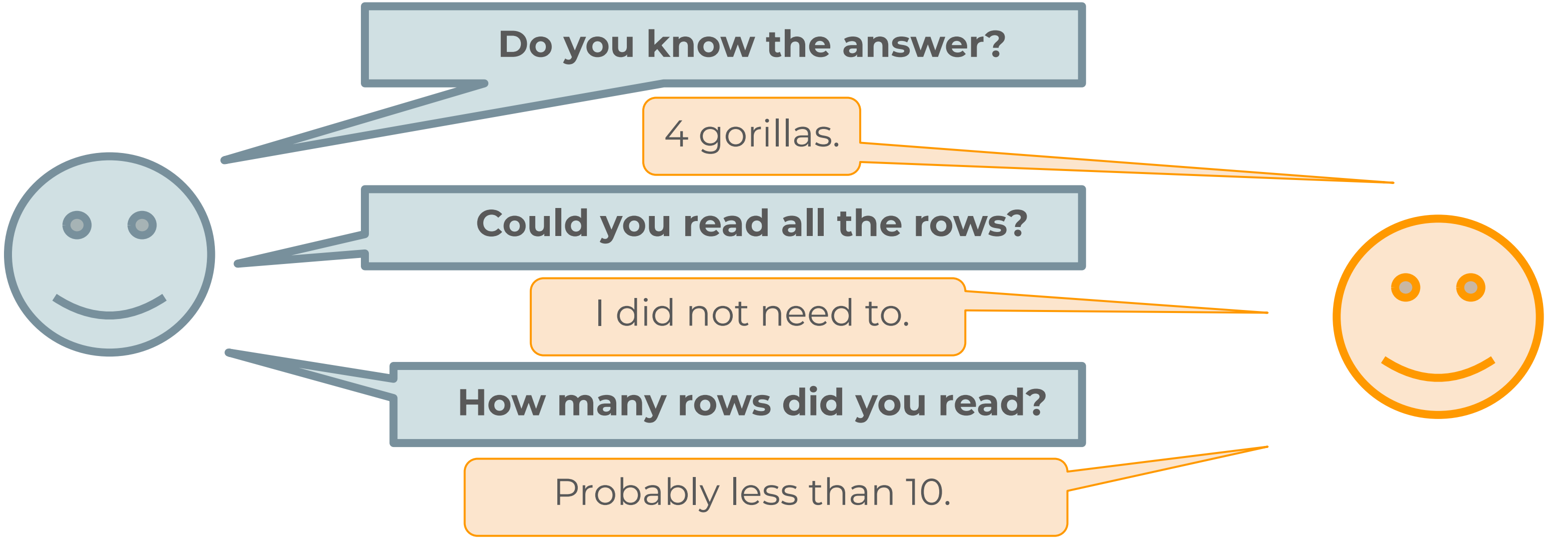
Searching Data

Name	Team
Johan	Black
Carmen	Black
Gemma	Black
Samantha	Black
Jennifer	Black
Alicia	Gorilla
Charles	Gorilla
Julia	Gorilla
George	Gorilla
Allison	White
Daniela	White
Patrick	White
Ellen	White
Elliot	White
Sean	White
Michael	White
Martha	White

Searching Data



Searching Data



Searching Data

Test Conclusions.

1. Knowing the kind of operations that will need to be done most often, will help reduce the searching time.

2. Preparing the data in a way that does not require reading all rows, will help reduce the searching time.

Searching Data

Finding data may take different times depending on how the data is organized.

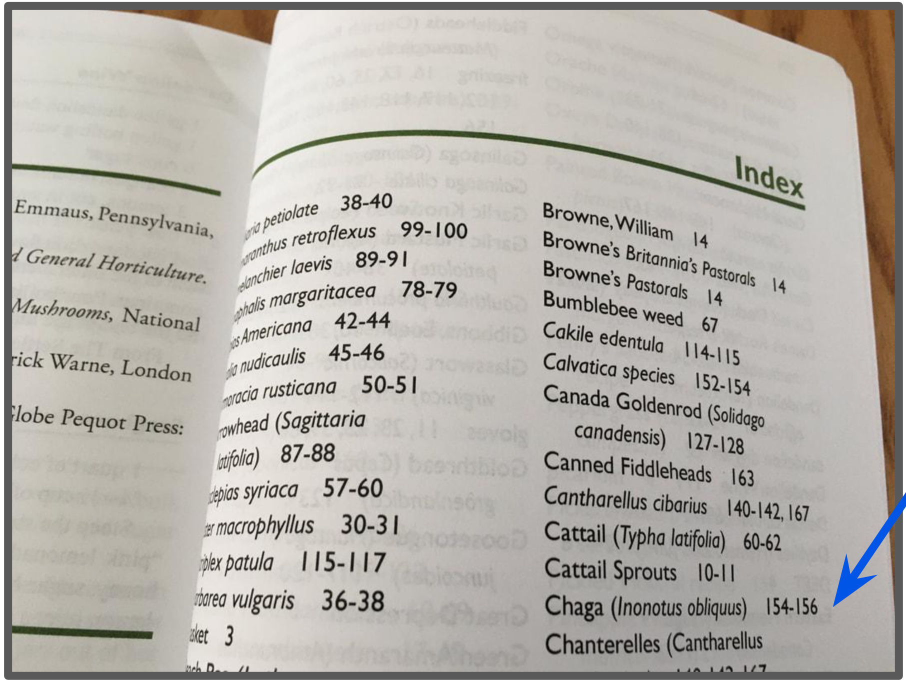
Photo by Oleksii Hlembotskyi on Unsplash

Photo by CHUTTERSNAP on Unsplash

Indexes

An **index** is a reference to the actual content.
It helps us search for specific information.

Instead of searching the entire book/table we **search the index**.



The index will prevent us from having to read 154 pages to find information about *Chaga*.

Indexes

Before computers, libraries used a system based on **indexed cards** to locate books.

An **index** is a list of references to the storage location of each element.

Photo by Maksym Kaharlytskyi on Unsplash

The index is sorted according to a specific field (year, author, title, ...).

Each index needs to be kept up to date to make the system useful.

Sequential vs. Indexed Scanning

Searching through all the rows is called **table scanning** or **sequential scanning**.

Name	Team
Allison	White
Alicia	Gorilla
Daniela	White
Charles	Gorilla
Johan	Black
Patrick	White
...	...

Searching through an indexed version of the data is called **indexed scanning**.

Name	Team
Johan	Black
Carmen	Black
Gemma	Black
Samantha	Black
Jennifer	Black
Alicia	Gorilla
...	...

If the search field is not indexed, the RDBMS can only do a table scan.
To be able to do an index scan, the field must have an index.

Database Indexes

A database **index** is a data structure used to improve the performance of search operations on specific fields.

Fact table				
Id	**Name**	**Last name**	**Team**	**...**
1	Allison	Müller	White	...
2	Alicia	Smith	Gorilla	...
3	Daniela	Hebbs	White	...
4	Charles	Bane	Gorilla	...
5	Johan	Schmidt	Black	...
6	Patrick	Peterson	White	...
...	...	...	...	...

Index table	
Team	**Id**
Black	5
Black	8
Black	11
Black	14
Black	16
Gorilla	2
...	...

- It is a different data structure that points to the data.
- An index works always on a specific field or combination of fields.

Brief Anatomy of Indexes

An **index** usually has two data: the searching field and a pointer to the actual data in storage.

Fact table

Id	Name	Last name	Team	...
1	Allison	Müller	White	...
2	Alicia	Smith	Gorilla	...
3	Daniela	Hebbs	White	...
4	Charles	Bane	Gorilla	...
5	Johan	Schmidt	Black	...
6	Patrick	Peterson	White	...
...	...	...	...	...

Index table

Team	Pointer
Black	CC470F...
Black	611189...
Black	E6894B...
Black	6079F9...
Black	FF18F1...
Gorilla	CC6654...
...	...

- Each row has an address in the RDBMS storage and the index table points to it.
- In some cases, indexes may instead hold the entire data.

Index Types

Primary indexes are defined on columns that guarantee there is one single record with the same value.

Fact table				
Id	**Name**	**Last name**	**Team**	**...**
1	Allison	Müller	White	...
2	Alicia	Smith	Gorilla	...
3	Daniela	Hebbs	White	...
4	Charles	Bane	Gorilla	...
5	Johan	Schmidt	Black	...
6	Patrick	Peterson	White	...
...	...	...	...	...

Index table	
Id	**Data block**
1	CC470F...
2	611189...
3	E6894B...
4	6079F9...
5	FF18F1...
6	CC6654...
...	...

The index table contains the indexed field and a pointer to the data record.

The index table is sorted by the indexed field.

Index Types

Secondary indexes are defined on columns that may have repeated values.

Fact table				
Id	**Name**	**Last name**	**Team**	**...**
1	Allison	Müller	White	...
2	Alicia	Smith	Gorilla	...
3	Daniela	Hebbs	White	...
4	Charles	Bane	Gorilla	...
5	Johan	Schmidt	Black	...
6	Patrick	Peterson	White	...
...	...	...	...	...

Index table	
Team	**Data block**
Black	CC470F...
Black	611189...
Black	E6894B...
Black	6079F9...
Black	FF18F1...
Gorilla	CC6654...
...	...

Create Indexes

The best way to create a **primary index** is to create a UNIQUE constraint.

```
CREATE TABLE people (
  id            integer PRIMARY KEY,
  first_name    varchar(20),
  last_name     varchar(50),
  social_sec    varchar(100) UNIQUE
);
```

PostgreSQL automatically creates a primary index when a `UNIQUE` constraint is created.

Defining a `PRIMARY KEY` automatically creates a `UNIQUE` constraint.

Create Indexes

```
personal=# \d people
                        Table "public.people"
   Column   |          Type          | Collation | Nullable | Default
------------+------------------------+-----------+----------+---------
 id         | integer                |           | not null |
 first_name | character varying(20)  |           |          |
 last_name  | character varying(50)  |           |          |
 social_sec | character varying(100) |           |          |
Indexes:
    "people_pkey" PRIMARY KEY, btree (id)
    "people_social_sec_key" UNIQUE CONSTRAINT, btree (social_sec)
```

Two indexes were created with the SQL of the previous slide.

Create Indexes

```
CREATE INDEX <index_name>
ON <table_name> (first_name);
```

An **index** can be manually created with SQL.

```
personal=# CREATE INDEX my_own_index ON people(last_name);
CREATE INDEX
```

An **index** can also be removed.

```
personal=# DROP INDEX my_own_index;
DROP INDEX
```

Create Indexes

```
personal=# CREATE INDEX ON people(last_name);
CREATE INDEX
personal=# \d people
                        Table "public.people"
   Column   |          Type          | Collation | Nullable | Default
------------+------------------------+-----------+----------+---------
 id         | integer                |           | not null |
 first_name | character varying(20)  |           |          |
 last_name  | character varying(50)  |           |          |
 social_sec | character varying(100) |           |          |
Indexes:
    "people_pkey" PRIMARY KEY, btree (id)
    "people_last_name_idx" btree (last_name)
    "people_social_sec_key" UNIQUE CONSTRAINT, btree (social_sec)
```

Omitting the index name will create an automated name.

Create Indexes

```
CREATE INDEX <index_name>
ON <table_name> USING <method>
(<column_name> DESC);
```

The method used to search the indexed key field can be specified with the `USING` construct.

By default, the method used is a self-balancing tree (B-Tree).

The values available for `<method>` are: btree, hash, gist & gin.

The sorting order of the indexed search field can be specified using `ASC` and `DESC`. By default it is sorted ascendingly.

Indexes Impact on Performance

The data structure must be created and maintained.

Fact table				
Id	**Name**	**Last name**	**Team**	**...**
1	Allison	Müller	White	...
2	Alicia	Smith	~~Gorilla~~Black	...
3	Daniela	Hebbs	White	...
4	Charles	Bane	Gorilla	...
5	Johan	Schmidt	Black	...
6	Patrick	Peterson	White	...
...	...	...	...	...

Index table	
Team	**Id**
Black	5
Black	8
Black	11
Black	14
Black	16
~~Gorilla~~Black	2
...	...

Any DML command on the table (`UPDATE`, `DELETE`, `INSERT`) will require updating the index and will add an overhead to these operations.

Indexes Impact on Performance

Defining indexes for every field
would make all the search operations fast.

Fact table

Id	Name	Last name	Team	...
1	Allison	Müller	White	...
2	Alicia	Smith	Gorilla	...
3	Daniela	Hebbs	White	...
4	Charles	Bane	Gorilla	...
5	Johan	Schmidt	Black	...
6	Patrick	Peterson	White	...
...	...	...	...	...

Index 1

Index 2

Index 3

Index 4

...

But would also multiply the overhead of any `UPDATE`, `DELETE` and `INSERT` operation.

Indexes Impact on Performance

Bigger tables will have bigger indexes that will take longer to build and maintain.

Fact table				
Id	**Name**	**Last name**	**Team**	**...**
1	Allison	Müller	White	...
2	Alicia	Smith	Gorilla	...
3	Daniela	Hebbs	White	...
4	Charles	Bane	Gorilla	...
...	...	...	...	...
999...	Patrick	Peterson	White	...
...	...	...	...	...

Index 1

Index 2

Index 3

Index 4

...

Having too many fields indexed on large tables may reduce dramatically the performance of DML operations on the table.

Database Workload

Database Workload

The workload on the database depends on two main factors:

Users	Complexity
Quantity	Quality

The **more operations** required per unit of time the higher the workload will be.

The **more complex** the operations requested the higher the workload will be.

The Complexity of SQL Queries

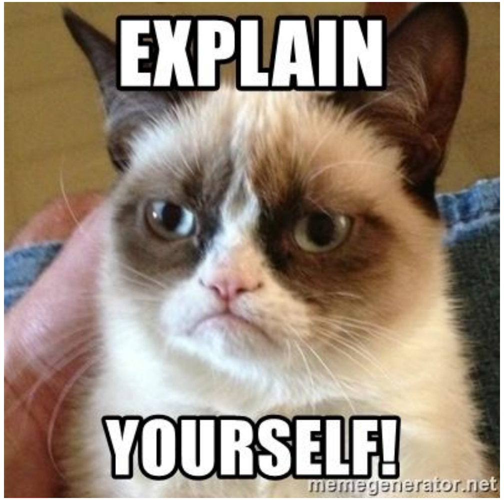
The PostgreSQL console provides a `\timing` command to measure the time spent on each operation.

```
personal=# \timing
Timing is on
personal=# SELECT * FROM people;
 id | first_name | last_name | social_sec
----+------------+-----------+------------
(0 rows)

Time: 46.376 ms
```

The Complexity of SQL Queries

The complexity of SQL queries can be explained using SQL itself.




```
EXPLAIN <SQL_Query>;
```

EXPLAIN

```
EXPLAIN <SQL_Query>;
```

```
personal=# EXPLAIN SELECT * FROM people;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on people  (cost=0.00..11.90 rows=190 width=398)
(1 row)
```

`EXPLAIN` does not execute the query, it just analyzes it.
This may be very useful with malformed queries that take too long to execute.

Reading EXPLAIN Output

DLI

```
personal=# EXPLAIN SELECT * FROM people;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on people  (cost=0.00..11.90 rows=190 width=398)
(1 row)
```

It indicates the type of scan used in the query (sequential or indexed).

It estimates various aspects of the performance.

Reading EXPLAIN Output

```
personal=# EXPLAIN SELECT * FROM people;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on people  (cost=0.00..11.90 rows=190 width=398)
(1 row)
```

Time spent preparing the data.

Total time spent.

Rows retrieved.

Bytes retrieved.

These values are just estimates. The final numbers can only be known when executing the query.

Getting the Final Numbers

```
personal=# EXPLAIN ANALYZE SELECT * FROM people;
                                   QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on people  (cost=0.00..11.90 rows=190 width=398)
                     (actual time=0.001..0.001 rows=0 loops=1)
 Planning time: 0.064 ms
 Execution time: 0.022 ms
(3 rows)
```

`EXPLAIN ANALYZE` will actually execute the query and provide the real time spent, as well as the final rows returned.

The `loops` value indicates the number of times the table (`people`) has been looped through. The values in `actual time` are for each loop, so the total time is this time multiplied by the number of loops.

Additional Information

```
personal=# EXPLAIN VERBOSE SELECT * FROM people;
                                 QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on public.people  (cost=0.00..11.90 rows=190 width=398)
   Output: id, first_name, last_name, social_sec
(2 rows)
```

`EXPLAIN VERBOSE` will add information on the schema used and the columns in the output.

`EXPLAIN ANALYZE VERBOSE` can also be used to combine all information.

Using Explain

```
personal=# EXPLAIN SELECT * FROM people WHERE first_name = 'Maria';
                       QUERY PLAN
----------------------------------------------------------
 Seq Scan on people  (cost=0.00..12.38 rows=1 width=398)
   Filter: ((first_name)::text = 'Maria'::text)
(2 rows)
```

If there is a **`WHERE`** clause but it uses a field without an index, the **`EXPLAIN`** output shows a sequential scan will be done.

The estimated total cost is slightly higher than using a query without a **`WHERE`** clause.

Using Explain

```
personal=# EXPLAIN SELECT * FROM people WHERE id = 1;
                                   QUERY PLAN
---------------------------------------------------------------------------
 Index Scan using people_pkey on people  (cost=0.14..8.16 rows=1 width=398)
   Index Cond: (id = 1)
(2 rows)
```

If there is a **`WHERE`** clause and it uses a field with an index, the **`EXPLAIN`** output will change.

The estimated total cost is 30% lower than searching a non-indexed column.

Using Explain

```
personal=# EXPLAIN SELECT * FROM people WHERE last_name = 'Smith';
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using people_last_name_idx on people  (cost=0.14..8.16 rows=1
width=398)
   Index Cond: (id = 1)
(2 rows)
```

A primary index and a secondary index show no performance difference.

Using Explain

```
map=# EXPLAIN SELECT * FROM city, country
map-# WHERE country.id = city.country_id AND code = 'DE';
                              QUERY PLAN
--------------------------------------------------------------------------
 Hash Join  (cost=8.18..22.04 rows=1 width=401)
   Hash Cond: (city.country_id = country.id)
   ->  Seq Scan on city  (cost=0.00..12.80 rows=280 width=259)
   ->  Hash  (cost=8.17..8.17 rows=1 width=142)
         ->  Index Scan using country_code_key on country
                                     (cost=0.15..8.17 rows=1 width=142)
               Index Cond: (code = 'ES'::bpchar)
(6 rows)
```

The first thing PostgreSQL will execute is the indexed filter condition in the country table. Then it will scan the city table sequentially, to combine those records with the cities.

Query Optimization

Populating Tables

1. Use a single `INSERT` of multiple rows instead of multiple `INSERT` commands.
2. If you use multiple `INSERT` commands, use a single transaction.
3. If possible, remove all indexes from the table before populating.
4. If possible, remove all foreign key constraints.

> If the database is in production and users are working on it, removing indexes and foreign key constraints may lead to poor performance and inconsistency.

Optimizing SQL Queries

Some of the most important issues to be optimized have to do with:

1. **Table size**. The larger the table the longer it will take to perform a sequential scan, even if it is only to retrieve one row.
2. **Joins**. If the `JOIN` statements return a lot of rows, the query may be slow.
3. **Aggregations**: Combining multiple rows to produce a result requires more computation than retrieving those rows.

Optimizing SQL Queries

Some tips:

1. **Be picky**. Select only the information you need, both in terms of rows (`WHERE`) and columns (`SELECT`). Use indexed columns to filter data whenever possible.
2. **Sort the joins**. Filter the first table before joining it to the rest. Make sure the initial table size is small, so later joins remain relatively small.
3. **Indexes**. Use indexes to do the first initial filter on the main table. Do not define indexes on columns not used often for searching.
4. **Aggregations**: If possible, avoid using aggregations on common queries with `JOIN` clauses.

Optimizing SQL Queries

Some more tips:

1. Do not use `JOIN` clauses on entire tables if they have many rows.
2. Use proper column types and limits (INT, BIGINT,...).
3. Run `EXPLAIN` on the most common or critical queries to identify issues. Especially when they use custom types of `JOIN` clauses (LEFT, RIGHT,...).
4. Use `JOIN` clauses instead of combining tables with the `FROM` clause. Check the order of the joins and filters used.
5. Do not combine the `IN` operator with subqueries returning big table sizes.

IN Subquery Example

```
SELECT * FROM view1 WHERE
id IN (1,2,3,4,5,6,7,8,9,10)
ORDER BY field1;
```

9ms

```
SELECT * FROM view1 WHERE
id IN (
  SELECT id FROM table ORDER BY field2 LIMIT 10
) ORDER BY field1;
```

25s

Application Optimization

Not only the queries must be optimized, but also the amount of queries requested must be efficient.

Applications must make an efficient use of each connection and reduce the number of queries required to obtain the same information.

A common malpractice is to perform **N+1 queries** to obtain the same information we could retrieve with a single query.

N+1 Queries

```
>>> cursor.execute("SELECT * FROM city")
>>> for city in cursor.fetchall():
...     cursor.execute(f"SELECT name FROM country WHERE id = {city[4]}")
...     country = cursor.fetchone()
...     print(f"City name: {city[1]}. Country name: {country[0]}")
...
City name: Berlin. Country name: Germany
City name: Marseille. Country name: France
```

If the table city has 100 rows, the above code will execute 101 queries. That is N+1 queries, N being the number of rows returned in the first query.

Each query is very simple and efficient, but adds an overhead in terms of network latency and database query planning that may be critical.

N+1 Queries

```
>>> cursor.execute("SELECT city.name, country.name "
...                "FROM city WHERE city.country_id = country.id")
>>> for city in cursor.fetchall():
...     print(f"City name: {city[0]}. Country name: {city[1]}")
...
City name: Berlin. Country name: Germany
City name: Marseille. Country name: France
```

The same information can be obtained using a single query that combines the appropriate tables.

N+1 Queries

The N+1 performance problem happens more often than expected, especially in complex queries, where it is easier for the developer to write multiple simple queries than writing a more complex (and efficient) one.

This is why it is important to understand the proper use of joins, indexes and checking the database logs to detect these kind of behaviors, as well as using `EXPLAIN` to better understand what happens in each query.

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute
DCI