Overview of Database Monitoring

Objectives

After watching this video, you will be able to:

- Explain why proactive monitoring is important
- Identify what baseline data you should create
- Describe the options available for monitoring database usage and performance

What is database monitoring?

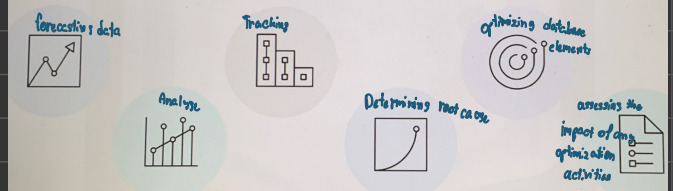
- Critical part of database management is database monitoring
- Scrutinization of day-to-day operational database status
- Crucial to maintain RDBMS health and performance

Why monitor your databases?

- Regular database monitoring helps identify issues in a timely manner
- If you do not monitor, database problems might go undetected
- RDBMSs offer tools to observe database state and track performance

Database monitoring tasks

Database admins use this information to perform several database monitoring tasks



Reactive monitoring

- After issue occurs – in direct response to the issue
- Typical scenarios include security breaches and critical performance levels

Proactive monitoring

- Prevents reactive panic by identifying issues before they grow larger
- Observes specific database metrics and sends alerts if values reach abnormal levels
- Uses automated processes
- Best strategy and preferred by most database admins

Establish a performance baseline

- Determines whether your database system is performing at its most optimal
- Record key performance metrics at regular intervals over a given time period

Establish a performance baseline

- Compare baseline statistics with database performance at any given time
- If measurements are significantly above or below baseline = analyze and investigate further

Establish a performance baseline

Use your performance baseline to determine operational norms:

- Peak and off-peak hours of operation
- Typical response times for running queries and processing batch commands
- Time taken to perform database backup and restore operations

Baseline data

The following areas typically have the greatest effect on the performance of your database system:

- System hardware resources
- Network architecture
- Operating system
- Database applications

Database monitoring options

Point-in-time (manual)

- Monitoring table functions
- Examine monitor elements and metrics
- Lightweight, high-speed monitoring infrastructure

Database monitoring options

Historical (automated)

- Event monitors
- Capture info on database operations
- Generate output in different formats

Monitoring Usage and Performance

Objectives

After watching this video, you will be able to:

- Explain why you need to monitor at different levels
- Describe the four levels at which you should monitor

Monitoring usage and performance

↳ Need appropriate tools

Need key performance indicators (KPIs) to measure database usage and performance

More commonly referred to as 'metrics'

Metrics enable DBAs to optimize organizations' databases for best performance

Regular monitoring also useful for operations, availability, and security

Monitoring at multiple levels

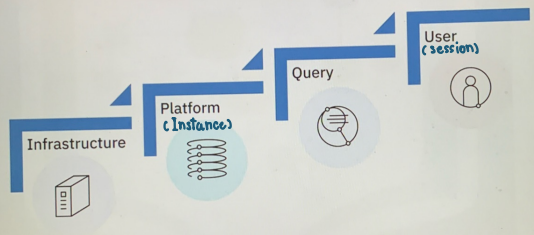
Ultimate goal of monitoring is to identify and prevent issues from adversely affecting database performance

Issues might be caused by:

- Hardware
- Software
- Network connections
- Queries
- Something else

Database monitoring should be multilevel

Monitoring at multiple levels



Monitoring at multiple levels

Underlying infrastructure components

- OS
- Servers
- Storage hardware
- Network components

↗ be working efficiently under the head of database platform and the queries

Infrastructure

Monitoring at multiple levels

Managing Db2, PostgreSQL, MySQL, or any other RDBMS a combination of more than one of these each platform is a consideration in terms of performance

Offers holistic insight into all elements necessary for consistent database performance

Platform

Monitoring at multiple levels

LOB apps repeatedly run queries against database

Most bottlenecks due to inefficient query statements:

- Cause latency
- Mishandle errors
- Diminish query throughput and concurrency

Query

Monitoring at multiple levels

Most misleading monitoring level because users are complaining about something not working, then you can just obtain further information about the issue they are having investigate it, but]

What if no users are complaining?

No complaints DOES NOT mean no issues

User/session

Monitoring at multiple levels

Successful monitoring happens continually and proactively:

- Monitoring *nirvana* is achieved when you identify issues before users are aware of them

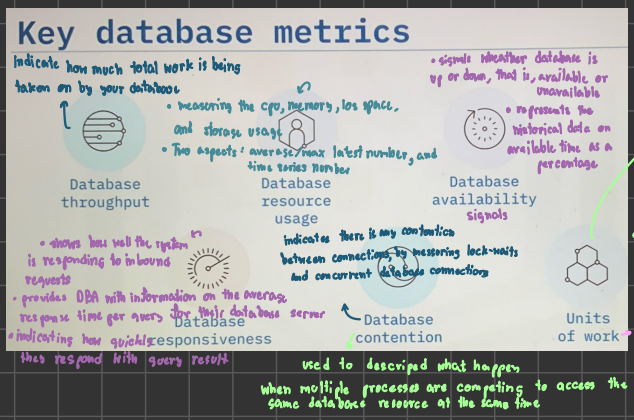
Monitoring at all levels is crucial to maintaining SLAs:

- High availability
- High uptime
- Low latency

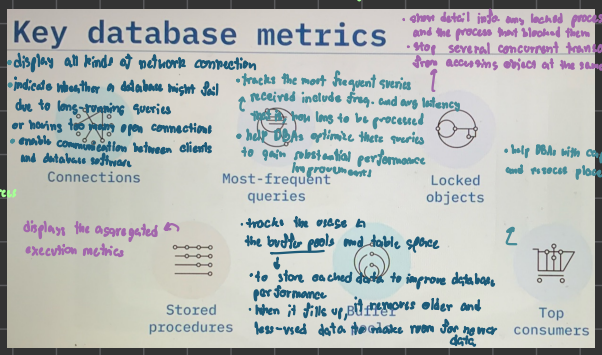
Objectives

After watching this video, you will be able to:

- Describe key database usage and performance metrics
- Describe the in-product tools for reviewing metrics
- Describe third-party monitoring tools



this is just a small selection of the numerous metrics available in most database management systems



Monitoring tools

Db2
- Db2 Data Management Console
- Workload manager
- Snapshot monitors

PostgreSQL
- pgAdmin dashboard (downloadable open-source tool)

MySQL
- MySQL Workbench: Performance Dashboard
- MySQL Workbench: Performance Reports
- MySQL Workbench: Query Statistics
- MySQL Query Profiler

Third-party monitoring tools

- pganalyze (PostgreSQL)
- PRTG Network Monitor (PostgreSQL, MySQL, SQL Server, Oracle)
- Available for multiple database systems:
  - SolarWinds Database Performance Analyzer
  - Quest Foglight for Databases
  - Datadog (database, system, and application monitoring)

Optimizing Database

Objectives

After watching this video, you will be able to:

- Describe why you need to optimize databases
- Describe the OPTIMIZE TABLE command for MySQL
- Describe the VACUUM and REINDEX commands for PostgreSQL
- Describe the RUNSTATS and REORG commands for Db2

Database optimization

Why do you need to optimize your databases?

- Identify bottlenecks
- Fine-tune queries
- Reduce response times

intro: over the time as the volume of data stored, data can be fragmented, table can be left empty

RDBMSs have their own optimization commands

- MySQL OPTIMIZE TABLE command
- PostgreSQL VACUUM and REINDEX commands
- Db2 RUNSTATS and REORG commands

MySQL OPTIMIZE TABLE command

- After significant amount of insert, update, or delete operations, databases can get fragmented
- OPTIMIZE TABLE reorganizes physical storage of table data and associated index to reduce storage space and improve efficiency
- Requires SELECT and INSERT privileges

```
OPTIMIZE TABLE accounts, employees, sales;
```

- Optimizes three tables in one operation
- You can also use phpMyAdmin graphical tool

PostgreSQL VACUUM command

- Garbage collection for PostgreSQL databases
  - Can also analyze (optional parameter)
- Reclaims lost storage space consumed by 'dead' tuples
- Regular use can help database optimization and performance
- Autovacuum does this for you (if enabled)

PostgreSQL VACUUM command

Examples:

```
VACUUM
```

- Frees up space on all tables

```
VACUUM tablename
```

→ create a complete copy of the table contents and write it to disk with no unused space

- Frees up space on specific table

```
VACUUM FULL tablename
```

require an exclusivity lock on each table

- Reclaims more space, locks database table, takes longer to run

PostgreSQL REINDEX command

→ REINDEX, it re-build the index contents from scratch

- Rebuild an index using the data stored in the index's table and replace the old version
- Must be owner of index, table, or database
- Reindexing has similar effect as dropping and recreating an index

```
REINDEX INDEX myindex;
```

- Rebuilds a single index

```
REINDEX TABLE mytable;
```

- Rebuilds all indexes on a table

Db2 RUNSTATS command

- Updates statistics in system catalog about characteristics of tables, associated indexes, or statistical views
  - Characteristics include # of records, # of pages, avg record length
- For a table, call RUNSTATS when table has had many updates, or after table is reorganized
- For a statistical view, call RUNSTATS when changes to underlying tables substantially affect rows returned by the view
  - View must be previously enabled for use in query optimization by using the ALTER VIEW statement

Db2 RUNSTATS command

Examples:

```
RUNSTATS ON TABLE employee
      WITH DISTRIBUTION ON COLUMNS (empid, empname)
```

- Collects stats on table only, with distribution stats on 'empid' and 'empname' columns

```
RUNSTATS ON TABLE employee for indexes empl1, empl2
```

- Collects stats on a set of indexes

Db2 REORG TABLE command

- Reorganizes a table by reconstructing rows to eliminate fragmented data and by compacting
- On a partitioned table, you can reorganize a single partition
- Affects all database partitions in database partition group

```
db2 REORG TABLE employee USE mytemp1
```

- Reorganizes a table to reclaim space and uses the temporary table space 'mytemp1'

Db2 REORG INDEX command

- Reorganize all indexes that are defined on a table by rebuilding the index data into unfragmented, physically contiguous pages
- Use CLEANUP option to perform cleanup without rebuilding indexes
- Affects all database partitions in database partition group

Optimizing Queries

IBM Developer

SKILLS NETWORK

Welcome to Optimizing Queries

Objectives

After watching this video, you will be able to:

- Explain how to use query execution plans
- Explain what query optimization is
- Describe some of the different query optimization tools available

IBM Developer SKILLS NETWORK

After watching this video, you will be able to:

- Explain how to use query execution plans,
- explain what query optimization is.
- and describe some of the different query optimization tools available.

Query execution plan

Steps used to access RDBMS data

- Shows details of a query execution plan for a statement

Obtaining query execution plan details:

- GUI tool
- Mode setting
- EXPLAIN statement

IBM Developer SKILLS NETWORK

A query execution plan (sometimes just referred to as a query plan) is the name given to the series of steps used to access data in RDBMSes when running statements.

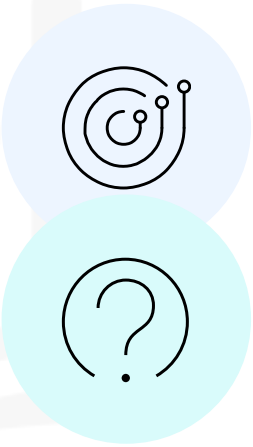
An RDBMS will often provide several methods for returning the details of a query execution plan.

Some RDBMSes offer tools which create a graphical representation of their query plans, while others allow a distinct mode to be set on the connection which causes the RDBMS to return a text-based description of their query plans.

And lastly, RDBMSes allow you to query a virtual database table, after running the query, by using an EXPLAIN statement.

Query optimization

- Query optimizer works out most efficient method for executing a query
- Evaluates possible query execution plans
- Database admins can manually fine - tune plans
- Some RDBMSes allow hints to be provided to query optimizer



IBM Developer SKILLS NETWORK

Most RDBMSes have a query optimization feature that uses a query optimizer tool to calculate the most efficient method for executing a query by evaluating all the available query execution plans.

When a query gets submitted to the database, the query optimizer evaluates the various possible query execution plans and returns what it determines to be the best choice.

However, query optimizers can be fallible, so database admins will sometimes need to manually inspect and fine-tune the plans produced by the query optimizer to get optimum query performance.

Some RDBMSes allow you to provide hints to the query optimizer. A hint is an additional component to the SQL statement that informs the database engine about how it wants it to execute a query, such as instructing the database engine to use an index when executing the query, even though the query optimizer might have decided not to.

EXPLAIN tools

- EXPLAIN statement
  - MySQL
  - PostgreSQL
  - Db2
- Graphical EXPLAIN tools
  - Db2 - Visual Explain
  - MySQL Workbench - Visual Explain Plan
  - PostgreSQL – PgAdmin graphical explain plan feature

IBM Developer SKILLS NETWORK

All flavors of RDBMSes, such as MySQL, PostgreSQL, and Db2 have an EXPLAIN statement that you can use to show a text-based representation of the details of a query execution plan for a statement, including the processes that occur and in what order they occur.

An EXPLAIN statement can be a good way to swiftly cure slow running queries.

Some RDBMSes also provide a graphical version of the EXPLAIN statement.

For example, Db2's Visual Explain uses information from a number of sources to enable you to view the access plan for explained SQL or XQuery statements as a graph. You can use the information available from the access plan graph to tune your queries for better performance.

For MySQL systems, the MySQL Workbench provides a Visual Explain Plan which produces and presents a visual representation of the MySQL EXPLAIN statement. MySQL Workbench provides all of the EXPLAIN formats for executed queries including the standard format, the raw extended JSON format, and the visual query plan.

And for PostgreSQL systems the **PgAdmin** utility provides a graphical explain plan feature. Although this is not an entire substitute for EXPLAIN or EXPLAIN ANALYZE text plans, it does offer a fast and simple method for viewing plans for additional analysis.

There also many third-party tools that can provide these same capabilities.

Summary

In this video, you learned that:

- Query execution plans show details of the steps used to access data when running query statements
- Most RDBMSes provide several methods for returning the details of a query execution plan
- Query optimization features use a query optimizer tool to determine the most efficient method for executing a query
- EXPLAIN statements show text -based details of a query execution plan for a statement

IBM Developer SKILLS NETWORK

In this video, you learned that:

- Query execution plans show details of the steps used to access data when running query statements.
- Most RDBMSes provide several methods for returning the details of a query execution plan, including EXPLAIN statements and visual explain tools.
- Query optimization features use a query optimizer tool to determine the most efficient method for executing a query.
- And EXPLAIN statements show a text-based representation of the details of a query execution plan for a statement.

Using Indexes

Objectives

After watching this video, you will be able to:

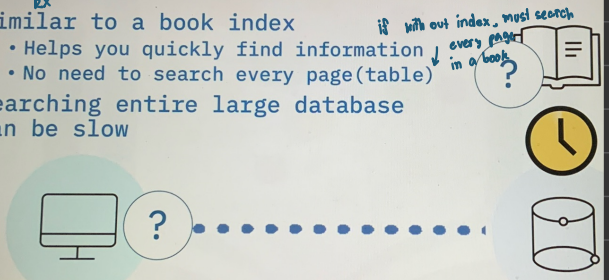
- Explain how indexing improves query performance
- Create effective indexes
- Describe different types of database indexes
- Create primary keys
- Create and drop database indexes
- List considerations for creating indexes

What is a database index?

Ex
Similar to a book index
- Helps you quickly find information
- No need to search every page(table)

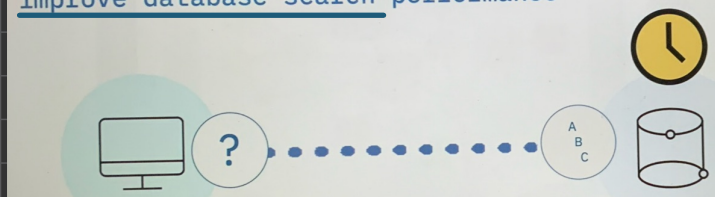
if without index, must search every page in a book

Searching entire large database can be slow



What is a database index?

A database index can significantly improve database search performance



What is a database index?

INDEX
11
11
11
12
15
18
18
20
20
20

ORDER_NO	CUST_ID	COST
33	11	45.02
36	15	26.11
39	12	66.26
44	20	15.47
48	11	92.01
49	18	103.50
53	11	89.13
56	20	46.55
61	18	29.17
63	20	40.22

- Ordered copy of selected columns of data
- Enables efficient searches without searching every row

What is a database index?

INDEX
11
11
11
12
15
18
18
20
20
20

ORDER_NO	CUST_ID	COST
33	11	45.02
36	15	26.11
39	12	66.26
44	20	15.47
48	11	92.01
49	18	103.50
53	11	89.13
56	20	46.55
61	18	29.17
63	20	40.22

such as

- Data columns defined by admins based on factors
  - Frequently searched terms, customer ID
- 'Lookup table' points to original rows in table

index serves

Creating effective indexes

find optimal balance and index maintenance

Ex.

Wide index
- may cover a greater number of queries
- require

Narrow index
- take up less disk space and require less overhead to maintain

need to be balanced

Performance
- or storage space

Maintenance
- they must be continually updated

Types of database indexes

- Primary key special type of index
  - Always unique, non-nullable, one per table
  - Clustered – data stored in table in order by primary key
- Indexes table can also have any number of additional indexes
  - Non-clustered, single or multiple columns
  - Unique or non-unique
- Column ordering is important
  - Ascending (default) or descending
  - First column first, then next, and so on

Creating primary keys

- Uniquely identifies each row in a table
- Good practice to create when creating the table

Syntax:

```
CREATE TABLE table_name
      (pk_column datatype NOT NULL PRIMARY KEY,
       column_name datatype,
       ...);
```

Example:

```
CREATE TABLE team
      (team_id INTEGER NOT NULL PRIMARY KEY,
       team_name VARCHAR(32));
```

Creating primary keys

Primary key with multiple columns

Syntax:

```
CREATE TABLE table_name
      (column_1_name datatype NOT NULL,
       column_2_name datatype NOT NULL,
       ...
       PRIMARY KEY(column_1_name, column_2_name));
```

Creating primary keys

Ensure uniqueness with auto-incrementing values

Db2: IDENTITY column

```
CREATE TABLE team
      (team_id INT GENERATED BY DEFAULT AS IDENTITY
                  PRIMARY KEY,
       team_name VARCHAR(32));
```

MySQL: AUTO_INCREMENT column

```
CREATE TABLE player
      ( player_id SMALLINT NOT NULL AUTO_INCREMENT,
        player name CHAR(30) NOT NULL,
        PRIMARY KEY (player_id) );
```

Creating indexes

Syntax:

```
CREATE INDEX index_name
      ON table_name (column_1_name, column_2_name, ...);
```

Examples:

```
CREATE UNIQUE INDEX unique_nam
      ON project (projname);
```

column

↖ this table can't have multiple rows with the same name as

```
CREATE INDEX job_by_dpt
      ON employee (workdept, job);
```

table can have multiple rows with the same

Dropping indexes

Syntax:

```
DROP INDEX index_name;
```

Example:

```
DROP INDEX job_by_dpt;
```

Primary key / unique key indexes cannot be explicitly dropped using this method

- Use ALTER TABLE statement instead

Considerations when creating indexes

- Major source of database bottlenecks is poorly designed or insufficient indexes

Database use?

Frequently used queries?

Column characteristics?

Index options?

Storage requirements?

Reading: Improving Performance of Slow Queries in MySQL

Estimated time needed: 20 minutes

In this reading, you'll learn how to improve the performance of slow queries in MySQL.

Objectives

After completing this reading, you will be able to:

1. Describe common reasons for slow queries in MySQL
2. Identify the reason for your query's performance with the `EXPLAIN` statement
3. Improve your query's performance with indexes and other best practices

Software Used

In this reading, you will see usage of MySQL. MySQL is a Relational Database Management System (RDBMS) designed to efficiently store, manipulate, and retrieve data.

Common Causes of Slow Queries

Sometimes when you run a query, you might notice that the output appears much slower than you expect it to, taking a few extra seconds, minutes or even hours to load. Why might that be happening?

There are many reasons for a slow query, but a few common ones include:

1. The size of the database, which is composed of the number of tables and the size of each table. The larger the table, the longer a query will take, particularly if you're performing scans of the entire table each time.

2. Unoptimized queries can lead to slower performance. For example, if you haven't properly indexed your database, the results of your queries will load much slower.

Each time you run a query, you'll see output similar to the following:

```
+--------+-----------+----------------+-----------------+--------+-----------+
300024 rows in set (0.34 sec)
```

As can be seen, the output includes the number of rows outputted and how long it took to execute, given in the format of `0.00` seconds.

One built-in tool that can be used to determine why your query might be taking a longer time to run is the `EXPLAIN` statement.

EXPLAIN Your Query's Performance

The `EXPLAIN` statement provides information about how MySQL executes your statement—that is, how MySQL plans on running your query. With `EXPLAIN`, you can check if your query is pulling more information than it needs to, resulting in a slower performance due to handling large amounts of data.

This statement works with `SELECT`, `DELETE`, `INSERT`, `REPLACE` and `UPDATE`. When run, it outputs a table that looks like the following:

```
mysql> EXPLAIN SELECT * FROM employees;
+----+-------------+-----------+------------+------+---------------+---
| id | select_type | table     | partitions | type | possible_keys | ke
+----+-------------+-----------+------------+------+---------------+---
|  1 | SIMPLE      | employees | NULL       | ALL  | NULL          | NU
+----+-------------+-----------+------------+------+---------------+---
1 row in set, 1 warning (0.00 sec)
```

As shown in the outputted table, with `SELECT`, the `EXPLAIN` statement tells you what type of select you performed, the table that select is being performed on, the number of rows examined, and any additional information.

In this case, the `EXPLAIN` statement showed us that the query performed a simple select (rather than, for example, a subquery or union select) and that 298,980 rows were examined (out of a total of about 300,024 rows).

The number of rows examined can be helpful when it comes to determining why a query is slow. For example, if you notice that your output is only 13 rows, but the query is examining about 300,000 rows—almost the entire table!—then that could be a reason for your query's slow performance.

In the earlier example, loading about 300,000 rows took less than a second to process, so that may not be a big concern with this database. However, that may not be the case with larger databases that can have up to a million rows in them.

One method of making these queries faster is by adding indexes to your table.

Indexing a Column

Think of indexes like bookmarks. Indexes point to specific rows, helping the query determine which rows match its conditions and quickly retrieves those results. With this process, the query avoids searching through the entire table and improves the performance of your query, particularly when you're using SELECT and WHERE clauses.

There are many types of indexes that you can add to your databases, with popular ones being regular indexes, primary indexes, unique indexes, full-text indexes and prefix indexes.

Type of Index	Description
Regular Index	An index where values do not have to be unique and can be NULL.
Primary Index	Primary indexes are automatically created for primary keys. All column values are unique and NULL values are not allowed.
Unique Index	An index where all column values are unique. Unlike the primary index, unique indexes can contain a NULL value.
Full-Text Index	An index used for searching through large amounts of text and can only be created for **char**, **varchar** and/or **text** datatype columns.
Prefix Index	An index that uses only the first N characters of a text value, which can improve performance as only those characters would need to be searched.

Now, you might be wondering: if indexes are so great, why don't we add them to each column?

Generally, it's best practice to avoid adding indexes to all your columns, only adding them to the ones that it may be helpful for, such as a column that is frequently accessed. While indexing can improve the performance of some queries, it can also slow down your inserts, updates and deletes because each index will need to be updated every time. Therefore, it's important to find the balance between the number of indexes and the speed of your queries.

In addition, indexes are less helpful for querying small tables or large tables where almost all the rows need to be examined. In the case where most rows need to be examined, it would be faster to read all those rows rather than using an index. As such, adding an index is dependent on your needs.

Be SELECTive With Columns

When possible, avoid selecting all columns from your table. With larger datasets, selecting all columns and displaying them can take much longer than selecting the one or two columns that you need.

For example, with a dataset of about 300,000 employee entries, the following query takes about 0.31 seconds to load:

```
1. 1
```

```
1. SELECT * FROM employee;
```

Copied!

```
| 499998 | 1956-09-05 | Patricia       | Breugel          | M      | 1993-10-13 |
| 499999 | 1958-05-01 | Sachin         | Tsukuda          | M      | 1997-11-30 |
+--------+------------+----------------+------------------+--------+------------+
300024 rows in set (0.31 sec)
```

But if we only wanted to see the employee numbers and their hire dates (2 out of the 6 columns) we could easily do so with this query that takes 0.12 seconds to load:

```
1. 1
```

```
1. SELECT employee_number, hire_date FROM employee;
```

Copied!

```
| 499998 | 1993-10-13 |
| 499999 | 1997-11-30 |
+--------+------------+
300024 rows in set (0.12 sec)
```

Notice how the execution time of the query is much faster compared to the when we selected them all. This method can be helpful when dealing with large datasets that you only need select specific columns from.

Avoid Leading Wildcards

Leading wildcards, which are wildcards (`"%abc"`) that find values that end with specific characters, result in full table scans, even with indexes in place.

If your query uses a leading wildcard and performs poorly, consider using a full-text index instead. This will improve the speed of your query while avoiding the need to search through every row.

Use the UNION ALL Clause

When using the `OR` operator with `LIKE` statements, a `UNION ALL` clause can improve the speed of your query, especially if the columns on both sides of the operator are indexed.

This improvement is due to the `OR` operator sometimes scanning the entire table and overlooking indexes, whereas the `UNION ALL` operator will apply them to the separate `SELECT` statements.

Next Steps

Congratulations! Now that you have a better understanding of why your query may be performing slow and how you can improve that performance, let's take a look at how we can do that with MySQL in the Skills Network Labs environment.

Improving Performance of Slow Queries in MySQL

6. With your password handy, we can now import the data. You can do this by entering the following into the Terminal:

```
mysql --host=mysql --port=3306 --user=root --password -t < employees.sql
```

When prompted, enter the password that was displayed under the **Connection Information** section when MySQL started up.

Please note, you won't be able to see your password when typing it in. Not to worry, this is expected!!

7. Your data will now load. This may take a minute or so.

When you've finished loading the data, you'll see the following:

```
theia@theiadocker-pratikshav:/home/project/employeesdb$ mysql --host=mysql --port=3306 --user=root --password -t < employees.sql
Enter password:
+-----------------------------+
| INFO                        |
+-----------------------------+
| CREATING DATABASE STRUCTURE |
+-----------------------------+
+-------------------------+
| INFO                    |
+-------------------------+
| storage engine: InnoDB  |
+-------------------------+
+---------------------+
| INFO                |
+---------------------+
| LOADING departments |
```

9. Recall that the name of the database that we're using is **Employees**. To access it, we can use this command:

```
use employees
```

```
mysql> use employees
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

10. Let's see which tables are available in this database:

```
show tables;
```

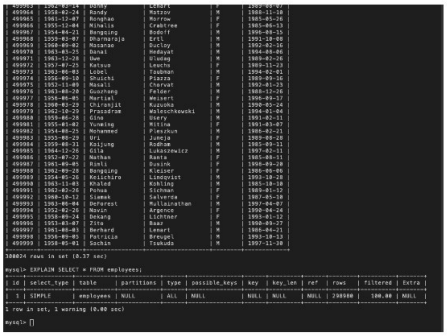
```
mysql> show tables;
+----------------------+
| Tables_in_employees  |
+----------------------+
| current_dept_emp     |
| departments          |
| dept_emp             |
| dept_emp_latest_date |
| dept_manager         |
| employees            |
| salaries             |
| titles               |
+----------------------+
8 rows in set (0.01 sec)

mysql>
```

Exercise 2: Check Your Query's Performance with EXPLAIN

The `EXPLAIN` statement, which provides information about how MySQL executes your statement, will offer you insight about the number of rows your query is planning on looking through. This statement can be helpful when your query is running slow. For example, is it running slow because it's scanning the entire table each time?

2. We can use `EXPLAIN` to see how many rows were scanned:

```
EXPLAIN SELECT * FROM employees;
```



Notice how `EXPLAIN` shows that it is examining 298,980 rows, almost the entire table! With a larger table, this could result in the query running slowly.

Exercise 3: Add an Index to Your Table

1. To begin, let's take at the existing indexes. We can do that by entering the following command:

```
SHOW INDEX FROM employees;
```

Remember that indexes for primary keys are created automatically, as we can see above. An index has already been created for the primary key, **emp_no**. If we think about this, this makes sense because each employee number is unique to the employee, with no NULL values.

2. Now, let's say we wanted to see all the information about employees who were hired on or after January 1, 2000. We can do that with the query:

```
SELECT * FROM employees WHERE hire_date >= '2000-01-01';
```

3. With the `EXPLAIN` statement, we can check how many rows this query is scanning:

```
EXPLAIN SELECT * FROM employees WHERE hire_date >= '2000-01-01';
```

This query results in a scan of 299,423 rows, which is nearly the entire table!

By adding an index to the **hire_date** column, we'll be able to reduce the query's need to search through every entry of the table, instead only searching through what it needs.

4. You can add an index with the following:

```
CREATE INDEX hire_date_index ON employees(hire_date);
```

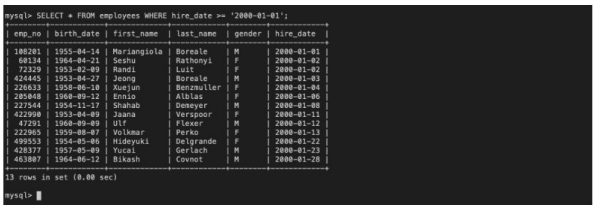
The `CREATE INDEX` command creates an index called **hire_date_index** on the table **employees** on column **hire_date**.

5. To check your index, you can use the `SHOW INDEX` command:

```
SHOW INDEX FROM employees;
```

6. Once more, let's select all the employees who were hired on or after January 1, 2000.

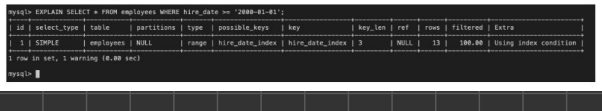
```
SELECT * FROM employees WHERE hire_date >= '2000-01-01';
```



The difference is quite evident! Rather than taking about 0.17 seconds to execute the query, it takes 0.00 seconds—almost no time at all.

7. We can use the `EXPLAIN` statement to see how many rows were scanned:

```
EXPLAIN SELECT * FROM employees WHERE hire_date >= '2000-01-01';
```



Under **rows**, we can see that only the necessary 13 columns were scanned, leading to the improved performance.

Under **Extra**, you can also see that it has been explicitly stated that the index was used, that index being **hire_date_index** based on the **possible_keys** column.

Now, if you want to remove the index, enter the following into the Terminal:

```
DROP INDEX hire_date_index ON employees;
```

This will remove the **hire_date_index** on the **employees** table. You can check with the `SHOW INDEX` command to confirm: