



Week 3 - Monitoring and Optimization

▼ Tipo	Anotaciones
☰ Posición	
⋮ Etiquetas	
↗ Bibliografía	
🕒 Fecha de creación	@June 13, 2022 7:09 PM
☰ Property	

▼ Monitoring and Optimization

▼ Overview of Database Monitoring

What is database monitoring?

- Critical part of database management
- 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

- Forecasting your future hardware requirements based on database usage patterns
- Analyzing the performance of individual applications or database queries
- Tracking the usage of indexes and tables.
- Determining the root cause of any system performance degradation
- Optimizing database elements to provide the best possible performance

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
- Proactive monitoring typically utilizes automated processes to perform tasks such as regularly verifying that a database system is online and accessible, verifying that configuration changes do not adversely affect the performance of the database system, and that the database system is operating and performing at acceptable levels
- Best strategy and preferred by most database admins

Establish a performance baseline

1. Determines whether your database system is performing at its most optimal
2. Record key performance metrics at regular intervals over a given time period
3. Compare baseline statistics with database performance at any given time
4. If measurements are significantly above or below baseline = analyze and investigate further

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
- Client applications

Database monitoring options

- Point-in-time (manual)
 - Monitoring table functions
 - Examine monitor elements and metrics
 - The monitoring table functions use a lightweight, high-speed monitoring infrastructure
- Historical (automated)
 - Event monitors
 - Capture info on database operations
 - Generate output in different formats

▼ Monitoring Usage and Performance - Part 1

Monitoring usage and performance

- 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

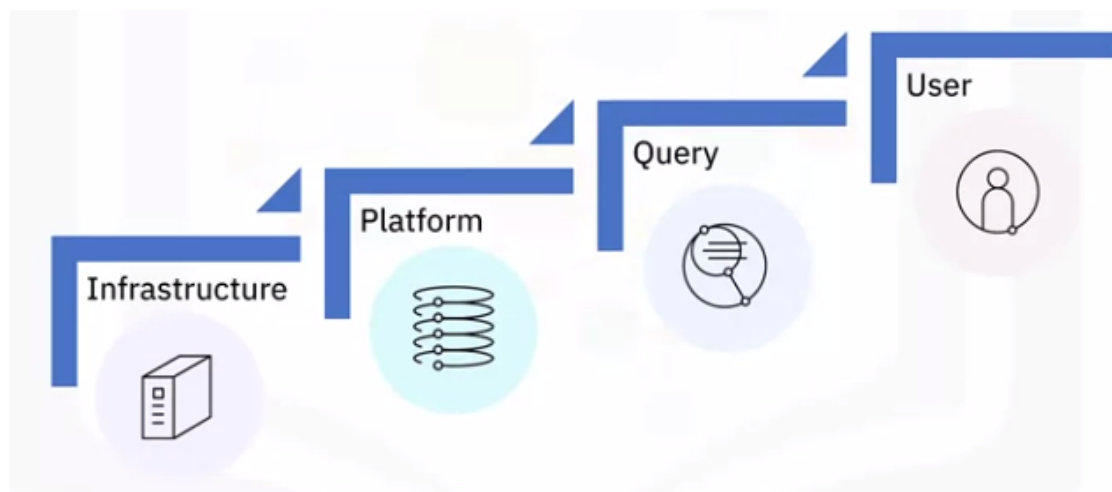
Monitoring at multiple levels

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:



Infrastructure

- OS
- Servers
- Storage hardware

- Network components

Platform

- Managing Db2, PostgreSQL, MySQL, or any other RDBMS
- Offers holistic insight into all elements necessary for consistent database performance

Query

- Line-of-business (LOB) applications repeatedly run queries against database
- Most bottlenecks due to inefficient query statements:
 - Cause latency
 - Mishandle errors
 - Diminish query throughput and concurrency

User/Session

This level can often be the most misleading monitoring level of them all, because if your users are complaining about something not working, then you can just obtain further information about the issue they are having, investigate it, and hopefully fix it. But what if your users aren't currently complaining about things not working? Can you just assume that everything is running fine, put your feet up, and relax? Unfortunately, no. Just because there may not be an issue right now, doesn't mean that there won't be one just around the corner.

Truly successful monitoring means constantly monitoring the usage, performance, and behavior of your database system in a proactive and continuous manner.

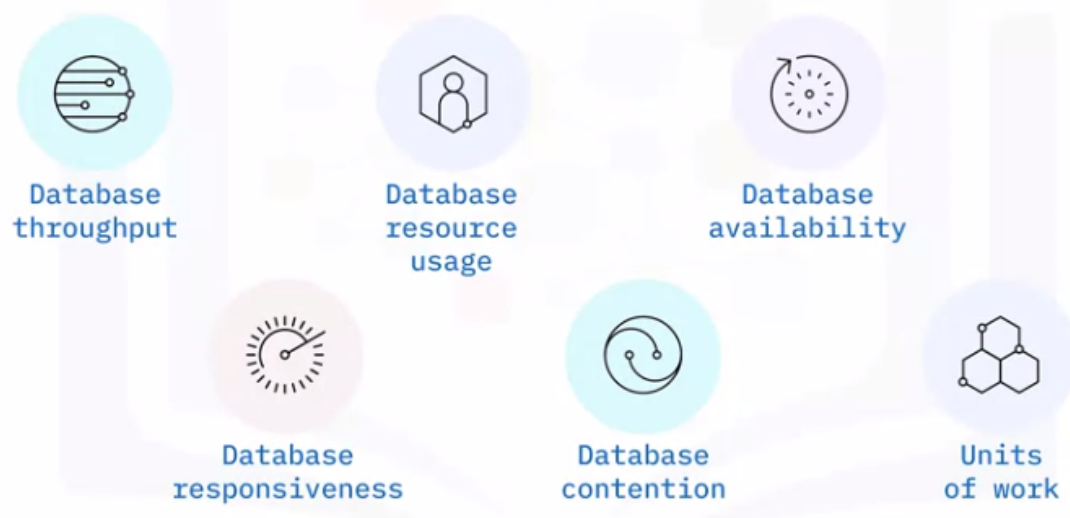
- The 'nirvana' of database monitoring is achieved when you proactively identify issues before your users are even aware of them.

Monitoring at all four of these levels is crucial to maintaining service level agreements (or SLAs), such as:

- High availability,
- High uptime
- Low latency for your databases.

▼ Monitoring Usage and Performance - Part 2

Key database metrics



Database throughput

Database throughput is one of the most significant metrics of database performance. It indicates how much total work is being taken on by your database and is typically measured by the number of queries executed per second.

Database resource usage

Database resource usage monitors the database resource usage by measuring the CPU, memory, log space, and storage usage. This summary metric represents the database resource usage by two aspects: average/max/latest number and time series number.

Database availability

Database availability signals whether the database is up or down, that is, available or unavailable. It is typically a summary metric that represents the historical data on available time as a percentage.

Database responsiveness

Database responsiveness shows how well the system is responding to inbound requests and is another of the more commonly used database performance

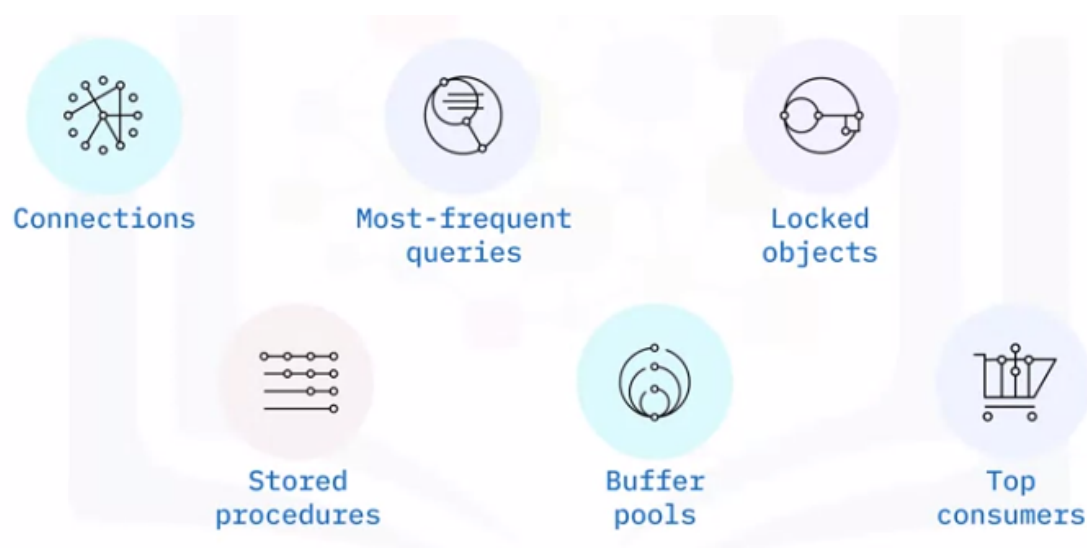
metrics. It provides DBAs with information on the average response time per query for their database servers, indicating how quickly they respond with query results.

Database contention

Database contention indicates whether there is any contention between connections, by measuring lock-waits and concurrent database connections. Database contention is the term used to describe what happens when multiple processes are competing to access the same database resource at the same time.

Units of work

Units of work tracks what transactions (units of work) are consuming the most resources on the database server.



Connections

Connections can display all kinds of network connection information to a database management console and can indicate whether a database server might fail due to long-running queries or having too many open connections. Database connections are network connections that enable communication between clients and database software. Open connections are used for sending commands and receiving responses in the form of result sets.

Most-frequent queries

Most frequent queries tracks the most frequent queries received by your database servers, including their frequency and average latency, that is, how long they take to be processed. It can help DBAs optimize these queries to gain substantial performance improvements.

Locket objects

Locked objects shows detailed information about any locked processes and the process that blocked them. Locks and blocks stop several concurrent transactions from accessing an object at the same time. They put contending processes on hold until that object has been released and is accessible again.

Store procedures

Stored procedures displays the aggregated execution metrics for procedures, external procedures, compiled functions, external functions, compiled triggers, and anonymous blocks invoked since database activation.

Buffer pools

Buffer pools tracks the usage of buffer pools and table spaces. A directory server uses buffer pools to store cached data and to improve database performance. When the buffer pool fills up, it removes older and less-used data to make room for newer data.

Top consumers

Top consumers shows the top consumers of a system's resources and can help DBAs with capacity planning and resource placement.

Monitoring tools

Db2

- Db2 Data Management Console
- Workload manager
- Snapshot monitors

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:
 - SolaWinds Database Performance Analyzer
 - Quest Foglight for Databases
 - Datadog (database, system, and application monitoring)

▼ Optimizing Databases

Database optimization

Why do you need to optimize your databases?

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

RDBMSs have their own optimization commands:

```
-- MySQL
OPTIMIZE TABLE
-- PostgreSQL
VACUUM
REINDEX
-- Db2
RUNSTATS
REORG
```

MySQL OPTIMIZE TABLE command

- If your MySQL database receives a significant number of insert, update, or delete operations, it can lead to fragmentation of your data files, meaning

that a lot of unused space is wasted, which in turn can impact the database's performance. Therefore, it is recommended that database admins defrag their MySQL tables on a regular basis.

- OPTIMIZE TABLE reorganizes physical storage of table data and associated index to reduce storage space and improve efficiency
- Requires SELECT and INSERT privileges
- Optimizes three tables in one operation:

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

- You can also use **phpMyAdmin** graphical tool

PostgreSQL VACUUM command

- The VACUUM command can be used on your PostgreSQL databases to perform garbage collection, and optionally, analysis tasks.
- VACUUM reclaims lost storage space consumed by 'dead' tuples, which are not physically removed from their database tables after being deleted or made obsolete by an update during routine PostgreSQL operations.
- Regularly reclaiming this lost storage space can help improve your databases' overall performance in PostgreSQL.
- If the autovacuum feature is enabled, then PostgreSQL will automate the vacuum maintenance process for you.

```
VACUUM
--> Frees up space on all tables
VACUUM tablename
--> Frees up space on specific table
VACUUM FULL tablename
--> Reclaims more space, locks database table, takes longer to run
```

Without the FULL parameter, the VACUUM command can run alongside normal read and write table operations, as it does not require an exclusivity lock, and any reclaimed space is retained for future reuse within the table it was vacuumed from. However, if you specify the FULL parameter, the command will create a

complete copy of the table contents and write it to disk with no unused space, which allows the space to be reclaimed by the operating system. So, using the FULL parameter can reclaim more space for you, but it takes much longer to run and requires an exclusivity lock on each table while it is being processed.

PostgreSQL REINDEX command

- Rebuild an index using the data stored in the index's table and replace the old version
 - It can be used when software bugs or hardware failures have corrupted an index, or when an index has become bloated and contains numerous empty, or nearly empty, pages.
- You need to be the owner of the index, or the table, or the database in order to reindex it.
- When you use REINDEX, it rebuilds the index contents from scratch, and therefore it has a very similar effect as dropping and recreating an index. However, the way that locks on reads and writes works on them is different.

```
REINDEX INDEX myindex;  
--> Rebuilds a single index  
REINDEX TABLE mytable;  
--> Rebuilds all indexes on a table
```

Db2 RUNSTATS commands

- Updates statistics in the system catalog about the characteristics of a table, associated indexes, or statistical views.
 - These characteristics include number of records, number of pages, and average record length. The optimizer uses these statistics to determine access paths to the data.
- For a table, call the RUNSTATS command when the table has had many updates, or after the table is reorganized. For tables, this command collects statistics for a table on the database partition from which it is invoked. If the table does not exist on that database partition, the first database partition in the database partition group is selected.

- For a statistical view, call the RUNSTATS command when changes to underlying tables substantially affect the rows that are returned by the view.
 - The view must be previously enabled for use in query optimization by using the ALTER VIEW statement. For views, this command collects statistics by using data from tables on all participating database partitions.

```
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 the rows to eliminate fragmented data, and by compacting information.
- On a partitioned table, you can reorganize a single partition.
- In terms of its scope, the REORG TABLE command affects all database partitions in the 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

- You can reorganize all indexes that are defined on a table by rebuilding the index data into unfragmented, physically contiguous pages.
 - On a data partitioned table, you can reorganize a specific nonpartitioned index on a partitioned table, or you can reorganize all the partitioned indexes on a specific data partition.
- If you specify the CLEANUP option of the index clause, cleanup is performed without rebuilding the indexes. You can clean up a specific index on a table, or you can clean up all of the indexes on the table.

- In terms of its scope, the REORG INDEX command affects all database partitions in the database partition group.

▼ Using Indexes

What is a database index?

Similar to a book index:

- Helps you quickly find information
- No need to search every page / table

If a computer searches a database without an index, it must search the entire database to find the information, which can be very slow—particularly in a very large database. A database index, however, can significantly improve search performance for a database.

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

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

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

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

In this example, we are selecting the customer ID column. Regardless of how they are constructed, the overall goal of an index is to help users find the information they need quickly and easily. So, an index serves as a 'lookup table' with a pointer or link to the original rows in the base table for which the index is created.

Creating effective indexes



While indexes can improve database performance, they also require additional storage space and maintenance because they must be continually updated. For any database, you will likely have to experiment with different types of indexes and indexing options. This will help you find an optimal balance of query performance and index maintenance—such as how much disk space and activity are required to keep the index up to date. For example, narrow indexes, or indexes with relatively few columns in the index key, take up less disk space and require less overhead to maintain, whereas wide indexes, while they may cover a greater number of queries, also require more space and maintenance.

Types of database indexes

- Primary key
 - Always unique, non-nullable, one per table
 - Clustered - data stored in table in order
- 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

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

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

-- Primary key with multiple columns
CREATE TABLE table_name
(column_1_name datatype NOT NULL,
 column_2_name datatype NOT NULL,
 ...
 PRIMARY KEY (column_1_name, column_2_name));

-- 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

```
CREATE INDEX index_name
ON table_name (column_1, column_1, ...);
```

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

This example creates a unique index named `unique_nam` in the `project` table, with a column named `projname`, meaning that the `project` table cannot have multiple rows with the same `projname` value. When `ON table_name` is specified, the `UNIQUE` option prevents the table from containing two or more rows with the same value of the index key.

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

The second example creates an index named `job_by_dpt` in the `employee` table, and it includes `workdept` and `job` columns in that order. This one is not unique, so the table can have multiple rows with the same `workdept` and `job` columns, but the benefit of the index is that query filtering on those fields will be faster than without an index.

Dropping indexes

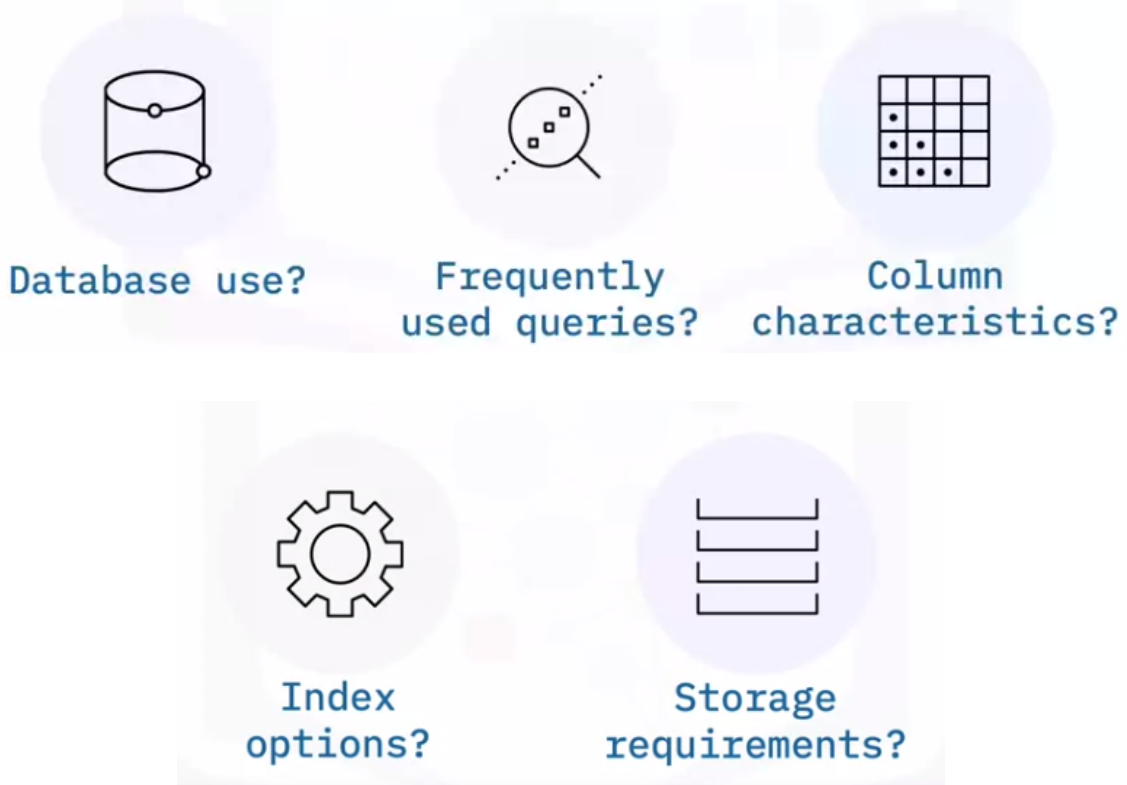
```
DROP INDEX index_name;  
  
DROP INDEX job_by_dpt;
```

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

- Use `ALTER TABLE` statement instead

Consideration when creating indexes

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



One of the major sources of database application bottlenecks can be poorly designed indexes, or a lack of sufficient, appropriate indexes. Therefore, designing efficient indexes is vital for optimizing database application performance. When designing an index, you should consider the following core principles:

- Understand how the database will be used. For example, will it be used for heavy online transaction processing or for the storage and retrieval of large data sets?
- Understand the most frequently used queries. For example, if you know that a frequently used query joins two or more tables, that may assist you in deciding the most appropriate index type to use.
- Understand the characteristics of the columns. For example, what kind of data will they contain? Will they contain integer values? And will they be unique? Knowing these characteristics from the outset will help you choose an appropriate index type.

- When creating or maintaining an index, which indexing options will best improve its performance? For example, if you are creating a clustered index on an existing large table, it would be a good idea to use the ONLINE index option, as it permits concurrent activity on the underlying data while the index is being built.
- And where is the best place to store the index? Where you decide to store your indexes can improve query performance by increasing the performance of disk input/output operations. For example, if you store a non-clustered index on a storage group located on a different disk than where the table storage group is located, it can improve performance as both disks could be read simultaneously.

▼ Reading: Improving Performance of Slow Queries in MySQL

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	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	NULL	ALL	NULL	NULL	NULL	NULL	298980	100.00	NULL

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.

<u>Aa</u> Type of Index	 Description
<u>Regular Index</u>	An index where values do not have to be unique and can be NULL.
<u>Primary Index</u>	Primary indexes are automatically created for primary keys. All column values are unique and NULL values are not allowed.
<u>Unique Index</u>	An index where all column values are unique. Unlike the primary index, unique indexes can contain a NULL value.
<u>Full-Text Index</u>	An index used for searching through large amounts of text and can only be created for char , varchar and/or text datatype columns.
<u>Prefix Index</u>	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:

```
SELECT *FROM employee;
```

```
| 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:

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

```
| 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.



Ungraded External Tool: Hands-on Lab: Improving Performance of Slow Queries in MySQL

Ungraded External Tool: Hands-on Lab: Monitoring and Optimizing your Databases in MySQL

Ungraded External Tool: Hands-on Lab: Monitoring and Optimizing your Databases in PostgreSQL