Week3_Optimizing_Databases

Monitoring and Optimization

- Critical part of database management
- Scruntinization of day-to-day operational database status
- Crucial to maintain RDBMS health and performance
- Regular database monitoring helps to 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 admins use this information to perform severral database monitoring tasks:
    - Forecast future hardware requirements based on database usage patterns
    - Analyze performance of individual applications or db queries
    - Tracking the usage of indexes and tables
    - Determine root cause of system performance issue or degradation
    - Optimizing database elements for performance

Reactive monitoring

- Done after issue occurs - in direct response to the issue
- Typical scenarios include security breaches and critical performance level

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 db 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 period
- Compare baseline statistics with database performance at any given time
- If measurements are significantly above or below baseline = analyze and investigate further
- Use your performance baseline to determine operational nors:
    - 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 monitor infrastructure

Historical (automated)

- event monitors
- capture info on database operations
- Generate output in different formats

Monitoring usage and performance

- Need key performance indicators (KPI) 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

Ultimate goal of monitoring is to identify and prevent isssues from adversely affecting database performance
Issues might be caused by:

- Hardware
- Software
- Network connections
- Queries
  Thus, database monitoring should be multilevel

Different levels:

- **Infrastructure**
  Underlying infrastructure components such as :
  - OS

    - Servers
    - Storage hardware
      Network components

- **Platform or database instance level**
  Managing DB2, PostgreSQL, MySQL or any other RDBMS
- Offers holistic insight into all elements necessary for consistent database performance
- **Query level**
  LOB apps repeatedly run queries against database
  Most bottlenecks due to inefficient query statements:
    - Cause latency
    - Mishandle errors
    - Diminish query throughput and concurrency
- **User/session level**
  Most misleading monitoring level
  No complaints DOES NOT MEAN no issues

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

---

Key database metrics

- Database throughput
- Database resource usage
- Database availability
- Database responsiveness
- Database contention
- Units of work
- Connections
- Most-frequent queries
- Locked objects
- Stored procedures
- Buffer pools
- Top consumers

Monitoring tools

- DB2
    - DB2 Data Management Console

    - Workload manager
    - Snapshot monitors
- PostgreSQL
    - pgAdmin dashboard
- 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, MSQL, SQL Server, Oracle)
    - Available for multiple database systems:
        - SolarWinds
        - Quest Foglight for Databases
        - Datadog (database, system, and application monitoring)

Optimizing Databases

Why to optimize?

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

RDBMS 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
`OPMIZE TTABLE` reorganizes physical storage of table data and associated index to reduce storge space and improve efficiency
Requires `SELECT` and `INSERT` privileges
`OPTIMZE TABLE accounts, employees, sales;`
You can use `phpAdmin` graphical tool as well.

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)

`VACUUM` - Frees up space on all tables

`VACUUM tablename` Frees up sapce on specific table

`VACUUM FULL tablename` Reclaims more space, locks database table, takes longer to run

PostgreSQL REINDEX command

- 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

Using Indexes

What is a database index?

Similar to a book index

- Helps you quickly find information
- No need to search every page (table)
- Searching entire large database can be slow, thus a database index can significantly improve search performance

```
Database index is an ordered copy of selected columns of data that enables
efficient searches without searching every row
```

Data columns defined by admins based on factors:

- Frequently searched terms, like customer ID
- "Lookup table" points to original rows in a table
- Can include one or more columns
- But they also require aditional space and maintenance

Types of database indexes

- **Primary Key**
  - Always unique, non-nullable, one per table
  - Clustedred-data stored in table in order by primary key
- **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 indexes

Syntax:

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

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

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

Dropping index

```
DROP INDEX index_name;
```

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
Understand:

- Database use?
- Frequently used queries?
- Column characteristics?
- Index options?
- Storage requirements?

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.

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`.

e.g: `EXPLAIN SELECT * FROM employees;`

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.

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.

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.

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.