Database Systems Lab

Lab # 05

Indexes and SQL Injection

Instructor: Engr. Muhammad Usman

Email: usman.rafiq@nu.edu.pk

Course Code: CL2005

Semester Fall 2021

Department of Computer Science,
National University of Computer and Emerging Sciences FAST
Peshawar Campus

Contents

5.1 Index

Indexes are **special lookup tables** that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discusses a certain topic, you first refer to the index, which lists all the topics alphabetically and are then referred to one or more specific page numbers.

An index helps to speed up **SELECT** queries and **WHERE** clauses, but it slows down data input, with the **UPDATE** and the **INSERT** statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the **CREATE INDEX** statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in an ascending or descending order.

Indexes can also be unique, like the **UNIQUE** constraint, in that the index prevents duplicate entries in the column or combination of columns on which there is an index.

The CREATE INDEX Command

The basic syntax of a **CREATE INDEX** is as follows.

```
CREATE INDEX index_name ON table_name;
```

Single-Column Indexes

A single-column index is created based on only one table column. The basic syntax is as follows.

```
CREATE INDEX index_name
ON table_name (column_name);
```

Unique Indexes

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows.

```
CREATE UNIQUE INDEX index_name
on table_name (column_name);
```

Composite Indexes

A composite index is an index on two or more columns of a table. Its basic syntax is as follows.

```
CREATE INDEX index_name
on table_name (column1, column2);
```

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

Implicit Indexes

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

The DROP INDEX Command

An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because the performance may either slow down or improve.

The basic syntax is as follows −

```
DROP INDEX index_name on table_name;
```

Displaying INDEX Information

You can use **SHOW INDEX** command to list out all the indexes associated with a table. Vertical format output specified by \G often is useful with this statement, to avoid long line wraparound:

```
Try out the following example:
SHOW INDEX FROM table_name\G
```

When should indexes be avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided.

The following guidelines indicate when the use of an index should be reconsidered.

- Indexes should not be used on small tables.
- Tables that have frequent, large batch updates or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed

5.2 SQL Injection:

SQL injection refers to the act of someone inserting a MySQL statement to be run on your database without your knowledge. Injection usually occurs when you ask a user for input, like their name, and instead of a name they give you a MySQL statement that you will unknowingly run on your database.

- SQL injection is a code injection technique that might destroy your database.
- SQL injection is one of the most common web hacking techniques.
- SQL injection is the placement of malicious code in SQL statements, via web page input.

Look at the following example which creates a SELECT statement by adding a variable ($txtUserId) to a select string. The variable is fetched from user input through POST method.

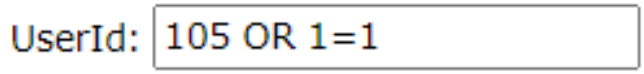
```
$txtUserId = $_POST["UserId"];
$txtSQL = "SELECT * FROM Users WHERE UserId =
$txtUserId";
```

The rest of this section describes the potential dangers of using user input in SQL statements.

SQL Injection Based on 1=1 is Always True

Look at the example above again. The original purpose of the code was to create an SQL statement to select a user, with a given user id.

If there is nothing to prevent a user from entering "wrong" input, the user can enter some "smart" input like this:



Then, the SQL statement will look like this:

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

The SQL above is valid and will return ALL rows from the "Users" table, since **OR 1=1** is always TRUE.

Does the example above look dangerous? What if the "Users" table contains names and passwords?

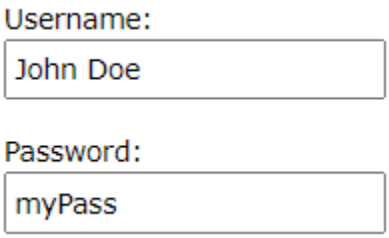
The SQL statement above is much the same as this:

```
SELECT UserId, Name, Password FROM Users WHERE UserId = 105 or 1=1;
```

A hacker might get access to all the user names and passwords in a database, by simply inserting 105 OR 1=1 into the input field.

SQL Injection Based on ""="" is Always True

Here is an example of a user login on a web site:



Example

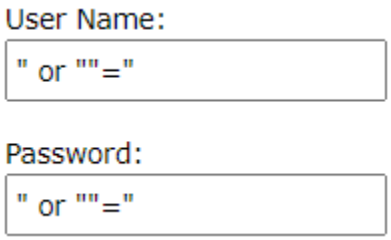
```
$uName = $_POST["username"];
$uPass = $_POST["userpassword"];

sql = “SELECT * FROM Users WHERE Name ='$uName’ AND Pass =' $uPass”
```

Result

```
SELECT * FROM Users WHERE Name ="John Doe" AND Pass ="myPass"
```

A hacker might get access to user names and passwords in a database by simply inserting " OR ""=" into the user name or password text box:



The code at the server will create a valid SQL statement like this:

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""
```

The SQL above is valid and will return all rows from the "Users" table, since **OR** ""="" is always TRUE.

SQL Injection Based on Batched SQL Statements

Most databases support batched SQL statement.

A batch of SQL statements is a group of two or more SQL statements, separated by semicolons.

The SQL statement below will return all rows from the "Users" table, then delete the "Suppliers" table.

Example

```
SELECT * FROM Users; DROP TABLE Suppliers
```

Look at the following example:

Example

```
$txtUserId = $_POST["UserId"];
$txtSQL = "SELECT * FROM Users WHERE UserId = $txtUserId";
```

And the following input:

User id: 105; DROP TABLE Suppliers

The valid SQL statement would look like this:

Result

```
SELECT * FROM Users WHERE UserId = 105; DROP TABLE Suppliers;
```

Use SQL Parameters for Protection

To protect a web site from SQL injection, you can use SQL parameters.

SQL parameters are values that are added to an SQL query at execution time, in a controlled manner.

The following examples shows how to build parameterized queries in PHP.

```
$user_id = $_POST[“User_id”];

$query_1 = $conn->prepare("select * from profile where user_id = ?");
$query_1->execute([$user_id]);
```

```
$user_id = $_POST[“User_id”];
$first_name = $_POST[“first_name”];
$last_name = $_POST[“last_name”];

$query_1 = $conn->prepare("insert into profile values(?,?,?)");
$query_1->execute([$user_id, $first_name, $last_name]);
```

```
$user_id = $_POST[“User_id”];
$first_name = $_POST[“first_name”];
$last_name = $_POST[“last_name”];

$query_1 = $conn->prepare("update profile set first_name = ?, last_name = ?
where user_id = ?");
$query_1->execute([$user_id, $first_name, $last_name]);
```

Parameterized queries can be used for any situation where untrusted input appears as data within the query, including the `WHERE` clause and values in an `INSERT` or `UPDATE` statement.