Complex Systems Theory and Practice

Lecture 4

Piotr Syga

2021-10-29

The basics

The idea

- 'saving' and 'ordering' of the values in a column (part of the column or multiple columns) in a given row, so we can avoid processing each of the rows
- the queries are made less time consuming, yet at the cost of memory usage
- the indexes are used automatically, if they can be used the optimizer decides to use them
- Note: similar at the first glance queries may be treated differently – most troublesome keywords: `LIMIT`, `LIKE` '%abc' vs 'abc%'

The basics

Creating an index

- There is always at least a single index in a table
- `CREATE INDEX` name_idx `USING` structure `ON` table (col1, col2);
- We can create an index on a part of a column, just as we limit the length of a char or varchar column.
- **Note:** Multicolumn indexes are using concatenation, hence the order is crucial. To use an index all caluses in the condition have to use indexes.

Structures: BTREE, RTREE (only some dialects), HASH
Note: We can select more options – a type of index (e.g., `FULLTEXT`, `SPATIAL`) or a creation algorithm (e.g., `INPLACE`, `COPY`)

Indexes – examples

Decide, in which of those queries index on the `name` column might be used:

```
SELECT * FROM Metals WHERE name LIKE 'go%';
SELECT * FROM Metals WHERE name LIKE '%ld';
SELECT * FROM Metals WHERE name <> 'steel' AND action='pull';
SELECT * FROM Metals JOIN Metals m ON Metals.action=m.action
WHERE m.name < Metals.name;
```

Indexes – costs

- memory overhead
- time overhead during `INSERT`s
- possible errors during table alteration and possibly 'dead' indexes (use `EXPLAIN` to verify)
- Halloween Problem

Indexes – costs

Halloween Problem

Increasing the pay for those earning under $25.000:

```
UPDATE EMPLOYEE
SET SALARY = SALARY * 1.1
WHERE SALARY<25000
```

What is the result?

Why there are no workers earning under $25.000 in the entire database?

Example of problem solution in T-SQL.

CTE

- Until now we 'saved' and reutilized queries by using views.
- Views allowed us to manage access control, limit the shown data and preserve backwards compability.
- Views may (and has to) be used just as a table (`SELECT`, `FROM`, `JOIN`, ...).

In the cases when we do not need access control and want to use a clearer and simpler solution (with additional perks) – we may use CTE.

CTE

- Until now we 'saved' and reutilized queries by using views.
- Views allowed us to manage access control, limit the shown data and preserve backwards compability.
- Views may (and has to) be used just as a table (`SELECT`, `FROM`, `JOIN`, ...).

In the cases when we do not need access control and want to use a clearer and simpler solution (with additional perks) – we may use CTE.

Common Table Expressions

- The constructions that allow us to 'save' a query to a specific kind of a variable.
- Does not require permissions to views or database modification.
- Allows the use of recursion.

CTE – definition and example

```
WITH (RECURSIVE)
name (columns) AS subquery
(name2 (columns) AS subquery2)
```

```
SET SESSION cte_max_recursion_depth = depth limit;
SET max_execution_time = time limit;
```

CTE – definition and example

```
WITH
cte1 AS (SELECT k1, k2 FROM Tab1),
cte2 AS (SELECT c1, c2 FROM Tab2)
SELECT k1, c1 FROM cte1 JOIN cte2
WHERE cte1.k2 > cte2.c2;
```

CTE – definition and example

```
WITH RECURSIVE fib (n, fib_n, next_fib_n) AS
(
SELECT 1, 0, 1
UNION ALL
SELECT n + 1, next_fib_n, fib_n + next_fib_n
FROM fib WHERE n < 13
)
SELECT * FROM fib;
SELECT fib_n FROM fib WHERE n=9;
```

Note: CTE can be used multiple times, however they exist only for the course of one query script.

Syntax of a basic loops

LOOP

label `LOOP`
commands
`END LOOP` *label*;

WHILE

(*label*) `WHILE`
commands
`END WHILE` (*label*);

Examples

Example of a loop

```
CREATE PROCEDURE ShaltThouCountToThree(c INT)
BEGIN
  countloop: LOOP
  IF c > 3 THEN
SET c = 0;
      ITERATE countloop;
ELSEIF c < 3 THEN
SET c = c + 1;
ITERATE countloop;
ELSE
LEAVE countloop;
    END IF;
  END LOOP countloop;
  SET @countNumber = c;
END;
```

Examples

Example of a while loop

```
CREATE PROCEDURE ThreeShaltBeTheNumberThouShaltCount(c INT)
BEGIN
  IF c > 3 THEN
SET c = 0;
    END IF;
WHILE c < 3 THEN
SET c = c + 1;
END WHILE;
  SET @countNumber = c;
END;
```

WHILE – another example

Let us assume table vendors(id, name, price, priority).

```
CREATE PROCEDURE ShoppingTime(money INT)
BEGIN
DECLARE shift INT DEFAULT 0;
DECLARE itPrice INT DEFAULT 0;
WHILE money > 0 THEN
 SELECT price INTO itPrice FROM vendors
ORDER BY priority LIMIT 1, shift;
IF money > itPrice THEN
SET shift = shift + 1;
END IF;
SET money=money-itPrice;
 SET @itemsBought = shift;
END;
```

Iterating over rows

- A constructions that lets us to iterate over a set of rows returned by a query or a procedure.
- We still remain unidirectional – no feedback loop.
- We can use them in functions, procedures or triggers.
- We cannot jump through the coursor – row by row.
- We cannot update tables' contents with coursors.
- Two types: asensitive (faster, directly on data) and insensitive (slower, working on copies).

Using a coursor

1. Creating:
   `DECLARE cursor_name CURSOR FOR (SELECT ...);`
   `DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;`
2. Initializing:
   `OPEN cursor_name;`
3. Getting the data:
   `FETCH cursor_name INTO (...);`
4. Cleaning up:
   `CLOSE cursor_name;`

Example

```
CREATE PROCEDURE p1(money INT)
BEGIN
  DECLARE done INT DEFAULT FALSE;
DECLARE cnt INT DEFAULT 0;
  DECLARE P,I INT;
DECLARE items varchar(300) DEFAULT "";
DECLARE itemCount CURSOR FOR
(SELECT price, name  FROM vendors ORDER BY priority);
DECLARE CONTINUE HANDLER
FOR NOT FOUND SET done = 1;
OPEN itemCount;
  read_loop: LOOP
    FETCH itemCount INTO P,I;
    IF done OR money < 0 THEN
      LEAVE read_loop;   END IF;
    IF money > P THEN
      SET cnt = cnt +1;
      SET items=CONCAT(items,"; ",item);
    END IF;
money = money - P;
  END LOOP;
CLOSE itemCount;
END;
```

Example 2

```
DELIMITER //
CREATE PROCEDURE p2()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE A VARCHAR(200);
  DECLARE N INT;
DECLARE ArtistDisco CURSOR FOR
(SELECT a.name, COUNT(t.TrackId) AS trackNO FROM artist a JOIN album aa
ON  a.ArtistId=aa.artistID JOIN track t ON t.albumID=aa.albumID
GROUP BY a.artistID);
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
OPEN ArtistDisco;
  read_loop: LOOP
    FETCH ArtistDisco INTO A,N;
    IF done THEN
      LEAVE read_loop;
    END IF;
    IF N > 15 THEN
      SELECT a.name, aa.title FROM artist a JOIN album aa ON
a.ArtistId=aa.artistID WHERE a.name=A;    END IF;
  END LOOP;
CLOSE ArtistDisco;
END; //
```

What we already know

- We know how to create database users
- We know how to grant different privileges to the users
- We know how to protect their password
- We know `PREPARE STATEMENT` keyphrase and can use it to validate the data
- We know how to log all the actions performed in a chosen table

What about large databases – does every client, every accountant and every manager require a separate DB user?
How to separate their data, how to protect them and how to produce authorization method for each separate user?

Real life scenario

- There are several **types** of end users, for each type (or selected few of the types) we create a DB user and grant respective privileges.
- The multiple users of the same type, the access is granted via frontend authorization – their credentials (username and a way to verify themselves) are stored in the database, in one of the tables.
- Since logging in requires reading data from that table, we do not want to keep the password as a plaintext.

PREPARE – a recall

Short example

```
SET @metal = 'steel';
SET @str = CONCAT("SELECT * FROM Metals WHERE name =",
@metal,";");
PREPARE stmt FROM @str;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
```

PREPARE – a recall

lazy evaluation

```
SET @metal = 'steel';
SET @str = CONCAT("SELECT * FROM Metals WHERE name =? ;");
PREPARE stmt FROM @str;
EXECUTE stmt USING @metal;
DEALLOCATE PREPARE stmt;
```

Idea

- a malicious user utilizes text fields to inject code (recall the last slide from Lecture 2)
- the main goals is to log in without credentials, learning the data in the database or modification of the database (incl. destroying the DB)
- the goal is achieved by preventing some parts of the code (queries) from running or changing their function
- the attacker may gain information and learn the structure of the database from the error messages

Simple authorization

Here we assume, for simplicity, that the password is given in plaintext – do not ever keep passwords in plaintext.

Fixed user, fixed password

```
SELECT userid FROM users WHERE name='Bob' AND password='BobbyB';
```

Simple authorization

Here we assume, for simplicity, that the password is given in plaintext – do not ever keep passwords in plaintext.

Fixed user, fixed password

```
SELECT userid FROM users WHERE name='Bob' AND password='BobbyB';
```

Using procedures

```
CREATE PROCEDURE auth (IN usr char(50), IN pass char(50), OUT ID
INT)
  SELECT userID INTO ID FROM users WHERE name=usr AND
password=pass;
```

How to use it: `CALL auth ('Bob','BobbyB',@uid)` – not very user friendly, hence we do move the quotation marks to the query.

Using concatenation

```
SET @str=CONCAT("SELECT userID FROM users WHERE name='",@usr,"' '
AND password='",@pass,"' ';");
```

How to log in?

- Without credentials of our own, we have to evaluate the `WHERE` clause (its condition) as true – normally it requires to find a row with both username and password just as the user typed in.
- Lets avoid normal way and try to evaluate the condition as TRUE in a different manner.
- Let us try both the name and password as a specific string: `' OR ''='`
- @str variable will look like
  "SELECT userID FROM users WHERE name='' OR ''='' AND password='' OR ''='';"
- Note that the condition is always true (even if the priorities are changed), hence the attacker will be loged in (if the database is not properly protected) as the first userID in the database.

More than loging in

What if first userID belong to some mock user with no privileges – can we learn the password of an arbitrary user?

- some functionalities may require confirmation with password
- the same credentials may be used on many websites
- we target a specific victim

If we previously logged in as an admin – we may just query the table of all users (DBE: `SELECT * FROM` mysql.user) or database (may vary due to design, usually `SELECT * FROM` [database].users).
However we may use bruteforce attack to win a guessing game determining the properties of the password:

- Does Bob's password contain 'B':
  Username - `U`
  Password - `' OR EXISTS(SELECT * FROM users WHERE name='Bob' AND password LIKE '%B%') AND ''='`
- Does it have 'b' after 'B': `...LIKE '%B%b%'...`
- Is 'o' the second character of the password: `...LIKE '_o%'...`