Structured Query Language

Advanced

EHSDI

eBuzima

Part 1 – Advanced Queries

Group By

- The **SQL GROUP BY** statement is used along with the SQL aggregate functions like SUM to provide means of grouping the result dataset by certain database table column(s).

Simplest Example:

- SELECT customer FROM orders GROUP BY customer;

- Rita
- Lucie
- Neza
- Marie Claire

NOTE : We could use DISTINCT to get the same result…So Why use GROUP BY???

The Power of GROUP BY

GROUP BY combined with aggregate functions

- SELECT customer, SUM(quantity) AS "Total Items" FROM orders GROUP BY customer;

- Rita 5
- Lucie 7
- Neza 3
- Marie Claire 6

What About No GROUP BY???

- SELECT customer,SUM(quantity) FROM orders

Rita	21
Lucie	21
Rita	21
Neza	21
Marie Claire	21
Neza	21
Neza	21

- NOTE: Here we have the total sum of the quantity column

Another Example

- SELECT customer, SUM((orders.quantity * inventory.price)) AS "COST" FROM orders JOIN inventory ON orders.product = inventory.product GROUP BY customer;

Notice the benefit of using a column alias

Also notice the math operations inside the sum

Multiple GROUP BY Example

- SELECT day_of_order, product, SUM(quantity) as "Total" FROM orders GROUP BY day_of_order,product ORDER BY day_of_order;

day_of_order	product	Total
2008-07-25 00:00:00.000	19" LCD Screen	5
2008-07-25 00:00:00.000	HP Printer	4
2008-08-01 00:00:00.000	Hanging Files	11
2008-08-01 00:00:00.000	Stapler	3
2008-08-15 00:00:00.000	19" LCD Screen	5
2008-08-16 00:00:00.000	Hanging Files	14

HAVING

- The SQL HAVING clause is like a WHERE clause for aggregated data.
- Any column name appearing in the HAVING clause must also appear in the GROUP BY clause.

HAVING Example

- SELECT day_of_order, product, SUM(quantity) as "Total" FROM orders GROUP BY day_of_order,product,quantity HAVING quantity > 7 ORDER BY day_of_order;

day_of_order **product** **Total**

- 2008-08-01 00:00:00.000 Hanging Files 11
- 2008-08-16 00:00:00.000 Hanging Files 14

Subqueries

- A query within another query. The "inner" query provides values for the criteria of the main query
- Joins are usually more efficient than subqueries

```
select * from crime where date in (select my_date from forecast_data);
```

id	date	crime	suspect_id
1	2009-06-28	armed robbery	1

Non-Correlated Subquery

- This subquery is non-correlated because it can be run on its own

```
select crime from crime where date in (select my_date from
forecast_data);
```

- In MySQL, try to use joins instead of non-correlated subqueries

```
select c.crime from crime c, forecast_data f where c.date =
f.my_date;
```

http://bugs.mysql.com/bug.php?id=9090

Correlated Subquery

- A correlated subquery is a query nested inside another query that uses values from the outer query in its WHERE clause.
- The sub-query is evaluated once for each row processed by the outer query.

```
SELECT employee_number, name FROM employee AS e1 WHERE salary > (SELECT
avg(salary) FROM employee WHERE department = e1.department);
```

- NOTE: Department average salary must be calculated every time since each employee could be in a different department.

Subqueries

- They allow queries that are structured so that it is possible to isolate each part of a statement.
- They provide alternative ways to perform operations that would otherwise require complex joins and unions.
- They are, in many people's opinion, more readable than complex joins or unions.

Joins

- In general, joins are faster. Many db optimizers convert subqueries into joins where possible.
- Joins are advantageous over sub-queries if the SELECT query contains columns from more than one table.
- For MySQL, it is especially true that you should use joins where possible because non-corralated subqueries are not optimized correctly.

Part 2 – DB Administration

CREATE TABLE

- Creates a table and sets defaults to null except for the id field

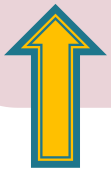
```
CREATE TABLE forecast_data2 (
        my_date date default NULL,
        description varchar(50) default NULL,
        details varchar(50) default NULL,
        id int
);
```

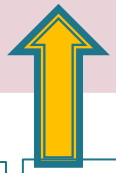
Create Table

```
mysql> show create table forecast_data;
+---------------+-----------------------------------------
| Table         | Create Table
+---------------+-----------------------------------------
| forecast_data | CREATE TABLE `forecast_data` (
  `my_date` date default NULL,
  `description` varchar(50) default NULL,
  `details` varchar(50) default NULL,
  `id` int(10) unsigned NOT NULL auto_increment,
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=12 DEFAULT CHARSET=latin1 |
+---------------+-----------------------------------------
```

Change the Name of a Column

```
alter table suspect
change column
id sid int not null auto_increment;
```





Old name

New name

ALTER

- alter table x add column b int;
- alter table x drop column b;
- alter table x modify column a varchar(50);
- alter table x modify column a a_new varchar(20);

Altering an Existing Column to Be a Primary Key

```
alter table suspect
modify column
id int not null auto_increment primary
key;
```

Constraints

- Restrict the type of data that can be entered into a table
  - Not Null
  - Unique
  - Primary Key
  - Foreign Key
  - Check
  - Default

UNIQUE

- Guarantees Uniqueness

```
alter table forecast_data
add column
(idu int unique);
```

Primary Key

- Constraint which serves as the identifier for each record in the table
- Each table should only have one primary key (may be a composite key – check out this interesting article and read the comments!

http://weblogs.sqlteam.com/jeffs/archive/2007/08/23/composite_primary_keys.aspx)

```
mysql> show create table forecast_data;
+---------------+-----------------------------------------
| Table         | Create Table
+---------------+-----------------------------------------
| forecast_data | CREATE TABLE `forecast_data` (
  `my_date` date default NULL,
  `description` varchar(50) default NULL,
  `details` varchar(50) default NULL,
  `id` int(10) unsigned NOT NULL auto_increment,
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=12 DEFAULT CHARSET=latin1 |
+---------------+-----------------------------------------
```

EHSDI
eBuzima

Foreign Key

- A foreign key refers to a primary key in another table
- The constraint maintains "referential integrity".
- This means it ensures the foreign key refers to a primary key that actually exists

Foreign Key

```
CREATE TABLE Orders
(
O_Id int NOT NULL,
OrderNo int NOT NULL,
P_Id int,
PRIMARY KEY (O_Id),
FOREIGN KEY (P_Id) REFERENCES Persons(P_Id)
)
```

CHECK

- Ensure the value inserted into the table conforms to a condition specified by CHECK

```
alter table forecast_data
add
column (idu int check(idu>0));
```

AUTO_INCREMENT

- Ensures that values for the column will automatically be incremented by one upon insert

```
alter table houses modify id int not null auto_increment;
```

Part 3 – DB Optimization

- http://20bits.com/articles/10-tips-for-optimizing-mysql-queries-that-dont-suck/
- http://dev.mysql.com/doc/refman/5.0/en/optimization.html
- http://www.smart-soft.co.uk/Oracle/oracle-performance-tuning-part1.htm

References

- http://dev.mysql.com/doc/
- http://www.tizag.com/sqlTutorial/sqlgroupby.php
- http://www.w3schools.com/sql/sql_groupby.asp
- http://www.w3schools.com/sql/sql_having.asp
- http://www.sql-tutorial.net/SQL-GROUP-BY.asp
- http://www.sql-tutorial.net/SQL-HAVING.asp

- Note: The official MySQL website is a great resource. Unlike many sites…yeah Tomcat…I'm looking at you…this website is very thorough and easy to understand. Even the search feature is well done!