# پوهنتون کابل

# پوهنځی کمپیوترساینس

## دیپارتمنت سیستم های معلوماتی

# Structured Query Language (SQL) Fundamentals

# Lectures 17-22

تهیه کننده : پوهنیار محمد شعیب "زرین خیل"
سال : 1389

# Structured Query Language (SQL) 17

By: M Shuaib Zarinkhail 2010

# SQL for Relational Algebra

- When DBs created, tables created, PK & FK assigned, RIC implemented …
- We need to retrieve data from a DB
- To do so we have to query as:
  - SELECT ColumnNames<br>FROM TableNames<br>WHERE condition (optional)
  - e.g. →select department, maxhours<br>from project;

# SELECT – Syntax

```
SELECT
[ALL | DISTINCT]
[HIGH_PRIORITY]
select_expr [, select_expr ...]
FROM table_references
[WHERE where_condition]
[GROUP BY {col_name | expr | position}
{ASC | DESC}, ... ]
```

*NEXT SLIDE CONTINUES*

# SELECT – Syntax

*… CONTINUES FROM PRECEDING SLIDE*

```
[HAVING where_condition]
[ORDER BY {col_name | expr | position}
{ASC | DESC}, ...]
[LIMIT row_count]
[INTO OUTFILE 'file_name' export_options
| INTO @var_name [, @var_name]]
```

# SELECT

- SELECT is used to retrieve rows chosen from one or more tables
- You can relate data from database objects:
- You can JOIN tables (two or more)
- You can use UNIONs and subqueries
- Support for UNION statements and subqueries is available from MySQL 4.0

# The most commonly used clauses of SELECT statements

- Each select_expr indicates a column that you want to retrieve
  - There must be at least one select_expr
- The table_references indicates the table or tables from which to retrieve records

# The most commonly used clauses of SELECT statements

- The WHERE clause indicates the conditions that rows must satisfy to be selected
  - where_condition is an expression that evaluates to true for each row to be selected
- The statement selects all rows if there is no WHERE clause
- In the WHERE clause, you can use any of the functions and operators that MySQL supports
  - Except for aggregate (summary) functions

# SELECT

- SELECT can also be used to retrieve rows computed without reference to any table
- For example:
  - SELECT 1 + 1;
- From MySQL 4.1.0 on, you are allowed to specify DUAL as a dummy table name in situations where no tables are referenced:
  - SELECT 1 + 1 FROM DUAL;

# SELECT

- DUAL is purely for the convenience of people who require that all SELECT statements should have FROM and possibly other clauses

- MySQL does not require FROM DUAL if no tables are referenced

# SELECT

Clauses used must be given in exactly the order shown in the syntax description

- For example, a HAVING clause must come after any GROUP BY clause and before any ORDER BY clause
- The exception is that the INTO clause can appear either as shown in the syntax description or immediately following the select_expr list

# SELECT –using * wildcard

- The list of select_expr terms comprises the select list that indicates which columns to retrieve
- Terms specify a column or expression or can use a * wildcard as shorthand:
  - i.e. select col1, col2, col3 from tOne;
  - i.e. select * from tOne;

# SELECT –using * wildcard

- A select list consisting only of a single unqualified * can be used as shorthand to select all columns from all tables:
  - SELECT * FROM t1 INNER JOIN t2 ...
- tbl_name.* can be used as a qualified shorthand to select all columns from the named table(s):
  - SELECT t1.*, t2.* FROM t1 INNER JOIN t2 ...

# Additional information about other SELECT Clauses

1: ALIAS names

- A select_expr can be given an alias using AS alias_name
  - The alias is used as the expression's column name and can be used in GROUP BY, ORDER BY, or HAVING clauses
  - For example:
  - SELECT CONCAT(last_name, ', ', first_name) AS full_name FROM mytable ORDER BY full_name;

# Additional information about other SELECT Clauses

2: ALIAS names

- The AS keyword is optional when aliasing a select_expr
  - The preceding example could have been written like this:
  - SELECT CONCAT(last_name, ', ', first_name) full_name FROM mytable ORDER BY full_name;

# Additional information about other SELECT Clauses

3: ALIAS names

- However, because the AS is optional, a subtle problem can occur if you forget the comma between two select_expr expressions:
  - MySQL interprets the second as an alias name
  - For example, in the following statement, columnB is treated as an alias name:
  - SELECT columnA columnB FROM mytable;

# Additional information about other SELECT Clauses

## 4: ALIAS names

- The FROM table_references clause indicates the table or tables from which to retrieve rows
  - If you name more than one table, you are performing a join
  - For each table specified, you can optionally specify an alias
  - tbl_name [[AS] alias]]

# Additional information about other SELECT Clauses

5: NAME REFERENCES

- You can refer to a table within the default database as tbl_name, or as db_name.tbl_name
- You can refer to a column as col_name, tbl_name.col_name, or db_name.tbl_name.col_name
- You need not specify a tbl_name or db_name.tbl_name prefix for a column reference unless the reference would be ambiguous

# Additional information about other SELECT Clauses

## 6: ALIAS names

- A table reference can be aliased using tbl_name AS alias_name or tbl_name alias_name:
  - SELECT t1.name, t2.salary FROM employee AS t1, info AS t2 WHERE t1.name = t2.name;
  - SELECT t1.name, t2.salary FROM employee t1, info t2 WHERE t1.name = t2.name;

# Additional information about other SELECT Clauses

7: COLUMN REFERENCES

- Columns selected for output can be referred to in ORDER BY and GROUP BY clauses using column names, column aliases, or column positions
  - Column positions are integers and begin with 1
  - The followings are equal queries:
  - SELECT college, region, seed FROM tournament ORDER BY region, seed;
  - SELECT college, region AS r, seed AS s FROM tournament ORDER BY r, s;
  - SELECT college, region, seed FROM tournament ORDER BY 2, 3;

# Additional information about other SELECT Clauses

8: COLUMN REFERENCES

- To sort in reverse order, add the DESC (descending) keyword to the name of the column in the ORDER BY clause

- The default is ascending order; this can be specified explicitly using the ASC keyword (optional)

# Additional information about other SELECT Clauses

## 9: COLUMN REFERENCES

- If ORDER BY occurs within a subquery and also is applied in the outer query, the outermost ORDER BY takes precedence
  - For example, results for the following statement are sorted in descending order, not ascending order:
  - (SELECT ... ORDER BY a) ORDER BY a DESC;

# Additional information about other SELECT Clauses

## 10: DUPLICATE COLUMN NAMES

- MySQL allows duplicate column names
  - There can be more than one select_expr with the same name
  - SELECT 12 AS a, a FROM t GROUP BY a;
  - In that statement, both columns have the name ‘a’
  - To ensure that the correct column is used for grouping, use different names for each select_expr

# Additional information about other SELECT Clauses

11: LIMIT

- The LIMIT clause can be used to constrain the number of rows returned by the SELECT statement
- LIMIT takes one or two numeric arguments, which must both be nonnegative integer constants

# Additional information about other SELECT Clauses

12: LIMIT

- With two arguments, the first argument specifies the offset of the first row to return, and the second specifies the maximum number of rows to return
  - The offset of the initial row is 0 (not 1):
  - SELECT * FROM tbl LIMIT 5,10; /* Retrieves rows 6-15*/

# Additional information about other SELECT Clauses

## 13: LIMIT

- To retrieve all rows from a certain offset up to the end of the result set, you can use some large number for the second parameter
  - This statement retrieves all rows from the 96th row to the last:
  - SELECT * FROM tbl LIMIT 95,18446744073709551615;

# Additional information about other SELECT Clauses

14: LIMIT

- With one argument, the value specifies the number of rows to return from the beginning of the result set:
  - SELECT * FROM tbl LIMIT 5; /* Retrieves the first 5 rows */
- In other words, LIMIT row_count is equivalent to LIMIT 0, row_count

# Additional information about other SELECT Clauses

15: Prepared Statements

- For prepared statements, you can use placeholders (supported as of MySQL 5.0.7)

The following statements will return one row from the tbl table:

- SET @a=1;
- PREPARE STMT1 FROM 'SELECT * FROM tbl LIMIT ?';
- EXECUTE STMT1 USING @a;
- DEALLOCATE PREPARE STMT1;

# Additional information about other SELECT Clauses

16: Prepared Statements

- The following statements will return the second to sixth row from the tbl table:
  - PREPARE STMT2 FROM 'SELECT * FROM tbl LIMIT ?, ?';
  - SET @skip=1; SET @numrows=5;
  - EXECUTE STMT2 USING @skip, @numrows;
  - DEALLOCATE PREPARE STMT2;

# Additional information about other SELECT Clauses

## 17: Prepared Statements

- This example shows how to create a prepared statement by using a string literal to supply the text of the statement:
  - PREPARE stmt3 FROM 'SELECT SQRT(POW(?, 2) + POW(?,2)) AS hypotenuse';
  - SET @a = 3; SET @b = 4;
  - EXECUTE stmt3 USING @a, @b;
  - DEALLOCATE PREPARE stmt3;

# Additional information about other SELECT Clauses

18: Prepared Statements

- The following example is similar to the previous, but supplies the text of the statement as a user variable:
  - SET @s = 'SELECT SQRT(POW(?,2) + POW(?, 2)) AS hypotenuse';
  - PREPARE stmt4 FROM @s; SET @a = 6; SET @b = 8; EXECUTE stmt4 USING @a, @b;
  - DEALLOCATE PREPARE stmt4;

# Additional information about other SELECT Clauses

19: LIMIT

- If LIMIT occurs within a subquery and also is applied in the outer query, the outermost LIMIT takes precedence
  - The following statement produces two rows, not one:
  - (SELECT ... LIMIT 1) LIMIT 2;

# Additional information about other SELECT Clauses

20. SELECT ... INTO OUTFILE

- The SELECT ... INTO OUTFILE 'file_name' form of SELECT writes the selected rows to a text file
  - SELECT * INTO OUTFILE “E:\SQL89\BKUP\tOneBKup.txt” FROM tOne;
- The file is created in the specified location
- file_name cannot be an existing file
- The SELECT ... INTO OUTFILE statement is intended primarily to let you very quickly dump a table to a text file

# Additional information about other SELECT Clauses

21. SELECT … INTO OUTFILE

- SELECT … INTO OUTFILE is the complement of LOAD DATA INFILE
- The LOAD DATA INFILE command can be implemented to enter unlimited data from an external text file
- The text file should be located in the home directory for MySQL or in the database folder which is created by the DBMS

# Additional information about other SELECT Clauses

22. SELECT … INTO OUTFILE

- For the first option, you can use the following command
  - LOAD DATA INFILE “/DataFileName.txt” INTO TABLE tableName
  - i.e. load data infile “/tOneBKup.txt” into table tOne;
- For the second option, you just need the file name in the command
  - LOAD DATA INFILE “DataFileName.txt” INTO TABLE tableName
  - i.e. load data infile “tOneBKup.txt” into table tOne;

# Structured Query Language (SQL) 18 + Lab 03

By: M Shuaib Zarinkhail 2010

# SELECT

- Following the SELECT keyword, you can use a number of options that affect the operation of the statement
- The ALL, DISTINCT, and DISTINCTROW options specify whether duplicate rows should be returned
- If none of these options are given, the default is ALL (all matching rows are returned)

# SELECT

- DISTINCT and DISTINCTROW are synonyms and specify removal of duplicate rows from the result set
- HIGH_PRIORITY, STRAIGHT_JOIN, and options beginning with SQL_ are MySQL extensions to standard SQL

# MySQL Extensions to Standard SQL

1. HIGH_PRIORITY

- HIGH_PRIORITY gives the SELECT higher priority than a statement that updates a table
- You should use this only for queries that are very fast and must be done at once
- HIGH_PRIORITY cannot be used with SELECT statements that are part of a UNION

# MySQL Extensions to Standard SQL

2. STRAIGHT_JOIN

- STRAIGHT_JOIN forces the optimizer to join the tables in the order in which they are listed in the FROM clause
- You can use this to speed up a query if the optimizer joins the tables in nonoptimal order
- STRAIGHT_JOIN also can be used in the table_ references list (JOIN Syntax)

# MySQL Extensions to Standard SQL

3. SQL_BIG_RESULT

- SQL_BIG_RESULT can be used with GROUP BY or DISTINCT to tell the optimizer that the result set has many rows
- In this case, MySQL directly uses disk-based temporary tables if needed, and prefers sorting to using a temporary table with a key on the GROUP BY elements

# MySQL Extensions to Standard SQL

4. SQL_BUFFER_RESULT

- SQL_BUFFER_RESULT forces the result to be put into a temporary table
- This helps MySQL free the table locks early and helps in cases where it takes a long time to send the result set to the client
- This option can be used only for top-level SELECT statements, not for subqueries or following UNION

# MySQL Extensions to Standard SQL

5. SQL_SMALL_RESULT

- SQL_SMALL_RESULT can be used with GROUP BY or DISTINCT to tell the optimizer that the result set is small
- In this case, MySQL uses fast temporary tables to store the resulting table instead of using sorting

# MySQL Extensions to Standard SQL

6. SQL_CALC_FOUND_ROWS

- SQL_CALC_FOUND_ROWS (available in MySQL 4.0.0 and up) tells MySQL to calculate how many rows there would be in the result set, disregarding any LIMIT clause
- The number of rows can then be retrieved with SELECT FOUND_ROWS() function

# MySQL Extensions to Standard SQL

7. SQL_CACHE and SQL_NO_CACHE

- The SQL_CACHE and SQL_NO_CACHE options affect caching of query results in the query cache
- SQL_CACHE tells MySQL to store the result in the query cache if it is cacheable and the value of the query_cache_type system variable is 2 or DEMAND
- SQL_NO_CACHE tells MySQL not to store the result in the query cache

# Lab 03 – Movies Database

In this lab you have to:

- Create new tables in the movies database
- Enter data to the new created tables

At the end of lab time:

- Record your answers and turn them to the lab instructor
- Email your database backup file created by the mysqldump command to your instructor

# Structured Query Language (SQL) 19

By: M Shuaib Zarinkhail 2010

# JOIN – Syntax

- MySQL supports the following JOIN syntaxes for the table_references part of SELECT statements
- It can also be used for the multiple-table DELETE and UPDATE statements
- table_references:
  - table_reference, table_reference
  - table_reference INNER JOIN table_reference [join_condition]
  - table_reference {LEFT|RIGHT} [OUTER] JOIN table_reference join_condition
  - table_reference NATURAL [{LEFT|RIGHT} [OUTER]] JOIN table_reference

# JOIN – Example

1. SELECT * FROM table1,table2 WHERE table1.id=table2.id;
2. SELECT * FROM table1 LEFT JOIN table2 ON table1.id=table2.id;
3. SELECT * FROM table1 LEFT JOIN table2 USING (id);
4. SELECT * FROM table1 LEFT JOIN table2 ON table1.id=table2.id LEFT JOIN table3 ON table2.id=table3.id;

# JOIN

- A table reference can be aliased using tbl_name AS alias_name or tbl_name alias_name:
  - SELECT t1.name, t2.salary FROM employee AS t1, info AS t2 WHERE t1.name = t2.name;

# JOIN

- If there is no matching row for the right table in the ON or USING part in a LEFT JOIN, a row with all columns set to NULL is used for the right table
- You can use this fact to find rows in a table that have no counterpart in another table:
  - SELECT left_tbl.* FROM left_tbl LEFT JOIN right_tbl ON left_tbl.id = right_tbl.id WHERE right_tbl.id IS NULL;

# JOIN

- RIGHT JOIN works analogously to LEFT JOIN
- To keep code portable across databases, it is recommended that you use LEFT JOIN instead of RIGHT JOIN

# Subquery – Syntax

- A subquery is a SELECT statement within another statement
- Here is an example of a subquery:
  - SELECT * FROM t1 WHERE column1 = (SELECT column1 FROM t2);
- In this example, SELECT * FROM t1 ... is the outer query and (SELECT column1 FROM t2) is the subquery

# Subquery – Syntax

- We say that the subquery is nested within the outer query
- Therefore, it is possible to nest subqueries within other subqueries
- It can continue to a considerable depth
- A subquery must always appear within parentheses

# The Main Advantages of 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 more readable than complex joins or unions

# Subquery – Example

```
DELETE FROM t1
WHERE s11 > ANY
  (SELECT COUNT(*) FROM t2
  WHERE NOT EXISTS
       (SELECT * FROM t3
       WHERE ROW(5*t2.s1,77)=
              (SELECT 50,11*s1 FROM t4 UNION
                     SELECT 50,77 FROM
                     (SELECT * FROM t5) AS t5)));
```

# Subquery for Comparison

- A subquery can contain any of the keywords
  - i.e. UNION (SELECT a FROM t2 WHERE a=11 AND B=2 ORDER BY a LIMIT 10);
- The most common use of a subquery is in the form:
  - non_subquery_operand comparison_operator (subquery)
- Where comparison_operator is one of these operators:

= > < >= <= <> != <=>

# Subquery for Comparison

- Here is an example of a common-form subquery comparison that you cannot do with a join
- It finds all the rows in table t1 for which the column1 value is equal to a maximum value in table t2:
  - SELECT * FROM t1 WHERE column1 = (SELECT MAX(column2) FROM t2);

# Subquery for Comparison

- This example again is impossible with a join because it involves aggregating for one of the tables
- It finds all rows in table t1 containing a value that occurs twice in a given column:
  - SELECT * FROM t1 AS t WHERE 2 = (SELECT COUNT(*) FROM t1 WHERE t1.id = t.id);

# Subqueries with ANY, IN, and SOME

Syntax:

2. operand comparison_operator ANY (subquery)

4. operand IN (subquery)

6. operand comparison_operator SOME (subquery)

# Subqueries with ANY, IN, and SOME

- The ANY keyword means “return TRUE if the comparison is TRUE for ANY of the values in the column that the subquery returns”
- For example:
  - SELECT s1 FROM t1 WHERE s1 > ANY (SELECT s1 FROM t2);
  - Suppose that there is a row in table t1 containing (10)
    - The expression is TRUE if table t2 contains (21,14,7) because there is a value 7 in t2 that is less than 10
    - The expression is FALSE if table t2 contains (20,10), or if table t2 is empty
    - The expression is unknown if table t2 contains (NULL,NULL,NULL)

# Subqueries with ANY, IN, and SOME

- When used with a subquery, the word IN is an alias for = ANY
- Thus, these two statements are the same:
  - SELECT s1 FROM t1 WHERE s1 = ANY (SELECT s1 FROM t2);
  - SELECT s1 FROM t1 WHERE s1 IN (SELECT s1 FROM t2);
- IN and = ANY are not synonyms when used with an expression list
  - IN can take an expression list, but = ANY cannot
- NOT IN is not an alias for <> ANY, but for <> ALL

# Subqueries with ANY, IN, and SOME

- The word SOME is an alias for ANY
- Thus, these two statements are the same:
  - SELECT s1 FROM t1 WHERE s1 <> ANY (SELECT s1 FROM t2);
  - SELECT s1 FROM t1 WHERE s1 <> SOME (SELECT s1 FROM t2);
- Use of the word SOME is rare

# Subqueries with ALL

Syntax:

- operand comparison_operator ALL (subquery)
- The word ALL, which must follow a comparison operator, means “return TRUE if the comparison is TRUE for ALL of the values in the column that the subquery returns”
- For example:
  - SELECT s1 FROM t1 WHERE s1 > ALL (SELECT s1 FROM t2);

*continues to the NEXT SLIDE*

# Subqueries with ALL

- Suppose that there is a row in table t1 containing (10)
  - The expression is TRUE if table t2 contains (-5,0,+5) because 10 is greater than all three values in t2
  - The expression is FALSE if table t2 contains (12,6,NULL,-100) because there is a single value 12 in table t2 that is greater than 10
  - The expression is unknown (that is, NULL) if table t2 contains (0,NULL,1)
  - Finally, if table t2 is empty, the result is TRUE

# Subqueries with ALL

- So the following statement is TRUE when table t2 is empty:
  - SELECT * FROM t1 WHERE 1 > ALL (SELECT s1 FROM t2);
- But this statement is NULL when table t2 is empty:
  - SELECT * FROM t1 WHERE 1 > (SELECT s1 FROM t2);
- In addition, the following statement is NULL when table t2 is empty:
  - SELECT * FROM t1 WHERE 1 > ALL (SELECT MAX(s1) FROM t2);

# Structured Query Language (SQL) 20

By: M Shuaib Zarinkhail 2010

# Subqueries – EXISTS & NOT EXISTS

- If a subquery returns any rows at all, EXISTS subquery is TRUE, and NOT EXISTS subquery is FALSE
- For example:
  - SELECT column1 FROM t1 WHERE EXISTS (SELECT * FROM t2);
- Traditionally, an EXISTS subquery starts with SELECT *, but it could begin with SELECT 5 or SELECT column1 or anything at all

# Subqueries – EXISTS & NOT EXISTS

- For the preceding example, if t2 contains any rows, even rows with nothing but NULL values, the EXISTS condition is TRUE
- The following are some more realistic examples using EXISTS and NOT EXISTS keywords in a subquery

# Subqueries – EXISTS & NOT EXISTS

What kind of store is present in one or more cities?

- SELECT DISTINCT store_type FROM stores WHERE EXISTS (SELECT * FROM cities_stores WHERE cities_stores.store_type = stores.store_type);

What kind of store is present in no cities?

- SELECT DISTINCT store_type FROM stores WHERE NOT EXISTS (SELECT * FROM cities_stores WHERE cities_stores.store_type = stores.store_type);

# Subqueries – EXISTS & NOT EXISTS

What kind of store is present in all cities?

- SELECT DISTINCT store_type FROM stores s1 WHERE NOT EXISTS (SELECT * FROM cities WHERE NOT EXISTS (SELECT * FROM cities_stores WHERE cities_stores.city = cities.city AND cities_stores.store_type = stores.store_type));

# Subqueries in FROM Clause

- Subqueries are legal in a SELECT statement's FROM clause
- The actual syntax is:
  - SELECT ... FROM (subquery) [AS] name ...
- The [AS] name clause is mandatory
  - Because every table in a FROM clause must have a name
- Any columns in the subquery select list must have unique names

# Subqueries in FROM Clause

- For the sake of illustration, assume that you have this table:
  - CREATE TABLE t1 (s1 INT, s2 CHAR(5), s3 FLOAT);
- Here is how to use a subquery in the FROM clause, using the example table:
  - INSERT INTO t1 VALUES (1,'1',1.0);
  - INSERT INTO t1 VALUES (2,'2',2.0);
  - SELECT sb1,sb2,sb3 FROM (SELECT s1 AS sb1, s2 AS sb2, s3*2 AS sb3 FROM t1) AS sb WHERE sb1 > 1;

# Subqueries in FROM Clause

Here is another example:

- Suppose that you want to know the average of a set of sums for a grouped table
- This does not work:
  - SELECT AVG(SUM(column1)) FROM t1 GROUP BY column1;
- However, this query provides the desired information:
  - SELECT AVG(sum_column1) FROM (SELECT SUM(column1) AS sum_column1 FROM t1 GROUP BY column1) AS t1;

# Some practical points in SELECT statement

- Use field names instead of * wildcard
- While using Aggregate Functions use the GROUP BY command
  - To make sure the function is worked
  - e.g. select name, count (*) from project group by department;
- Do not use the HAVING keyword with out GROUP BY command
  - e.g. select name, count (*) from project group by department having count(*) > 1;

# Some practical points in SELECT statement

- Make clear JOIN commands
  - to show which table is in the left side and which table is in the right side
- Use SQL-92 base JOIN commands
  - e.g. name every JOIN word as LEFT OUTER JOIN, RIGHT OUTER JOIN …
- While joining a table to its own, be careful (recursive)

# Some practical points in SELECT statement

- OUTER JOINs usually show NULL values
- Use care for the Boolean statement(s) after the WHERE keyword
  - It should be logically correct
- Run query for several times before using it in practical environment

# SELECT Statement – Examples PROJECT table

| ProjectID | Name | Department | MaxHours |
|---|---|---|---|
| 1000 | 03 Portfolio Analysis | Finance | 75.0 |
| 1200 | 03 Tax Prep | Accounting | 145.0 |
| 1400 | 04 Product Plan | Marketing | 138.0 |
| 1500 | 04 Portfolio Analysis | Finance | 110.0 |

# SELECT Statement – Examples EMPLOYEE table

| Employee Number | Name | Phone | Department |
|---|---|---|---|
| 100 | Mary Jacobs | 285-8879 | Accounting |
| 200 | Keni Numoto | 287-0098 | Marketing |
| 300 | Heather Jones | 287-9981 | Finance |
| 400 | Rosalie Jackson | 285-1273 | Accounting |
| 500 | James Nestor | 287-0123 | Info Systems |
| 600 | Richard Wu | 287-3222 | Info Systems |
| 700 | Kim Sung |  | Marketing |

# SELECT Statement – Examples ASSIGNMENT table

| ProjectID | EmployeeNum | HoursWorked |
|---|---|---|
| 1000 | 100 | 17.50 |
| 1000 | 300 | 12.50 |
| 1000 | 400 | 8.00 |
| 1000 | 500 | 20.25 |
| 1200 | 100 | 45.75 |
| 1200 | 400 | 70.50 |
| 1200 | 600 | 40.50 |
| 1400 | 200 | 75.00 |
| 1400 | 700 | 20.25 |
| 1400 | 500 | 25.25 |

# Structured Query Language (SQL) 21

By: M Shuaib Zarinkhail 2010

# SELECT Statement – Examples

- Reading some fields from one table (projection)

```
SELECT Name, Department, MaxHours
    FROM    PROJECT;
```

| Name | Department | MaxHours |
|---|---|---|
| Q3 Portfolio Analysis | Finance | 75.0 |
| Q3 Tax Prep | Accounting | 145.0 |
| Q4 Product Plan | Marketing | 138.0 |
| Q4 Portfolio Analysis | Finance | 110.0 |

# SELECT Statement – Examples

- The result of a query is always shown in one table
  - Even if it queries from multiple tables
  - In some cases a table with one row and zero records can be the result of a query
  - e.g. select Name from PROJECT where MaxHours > 150.0;

| Name |
|---|
| |

# SELECT Statement – Examples

- We can restructure a table by SELECT statement

```
SELECT     Name, MaxHours, Department
FROM         PROJECT;
```

| Name | MaxHours | Department |
|---|---|---|
| Q3 Portfolio Analysis | 75.0 | Finance |
| Q3 Tax Prep | 145.0 | Accounting |
| Q4 Product Plan | 138.0 | Marketing |
| Q4 Portfolio Analysis | 110.0 | Finance |

# SELECT Statement – Examples

- This query only shows one field

```
SELECT Department
    FROM    PROJECT;
```

- The first and last rows have the same (repeated) data
  - To eliminate it see next slid

| Department |
|---|
| Finance |
| Accounting |
| Marketing |
| Finance |

# SELECT Statement – Examples

- This query only shows one field (no repeated data)

```
SELECT DISTINCT Department
    FROM        PROJECT;
```

| **Department** |
|---|
| Finance |
| Accounting |
| Marketing |

# SELECT Statement – Examples

- Reading some rows from one table (selection)

```
  SELECT     ProjectID, Name, Department, MaxHours
FROM         PROJECT WHERE    Department =
  ‘Finance’;
```

| Project ID | Name | Department | MaxHours |
|---|---|---|---|
| 1000 | Q3 Portfolio Analysis | Finance | 75.0 |
| 1500 | Q4 Portfolio Analysis | Finance | 110.0 |

# SELECT Statement – Examples

- An alternative way to the previous query

```
SELECT *
   FROM PROJECT WHERE Department =
‘Finance’;
```

| ProjectID | Name | Department | MaxHours |
|---|---|---|---|
| 1000 | Q3 Portfolio Analysis | Finance | 75.0 |
| 1500 | Q4 Portfolio Analysis | Finance | 110.0 |

# SELECT Statement – Examples

- We can use more than one condition after WHERE

```
SELECT * FROM PROJECT
  WHERE Department=‘Finance’ AND
MaxHours>100;
```

| Project ID | Name | Department | MaxHours |
|---|---|---|---|
| 1500 | Q4 Portfolio Analysis | Finance | 110.0 |

# SELECT Statement – Examples

- We can use both selection & projection in one query

```
SELECT  Name, Department
    FROM       EMPLOYEE
    WHERE Department = ‘Accounting’;
```

| **Name** | **Department** |
|---|---|
| Mary Jacobs | Accounting |
| Rosalie Jackson | Accounting |

# WHERE Clause

Relational Operators / Descriptions used for queries:

| | |
|---|---|
| > | Greater Than |
| >= | Greater Than or Equal To |
| < | Less Than |
| <= | Less Than or Equal To |
| = | Equal To |
| <> | Not Equal To |
| != | Not Equal To |
| IN (list) | Contained in comma-separated list |
| LIKE string | Matches string pattern |

# WHERE Clause

- Logical Operators
- We use logical operators to combine the results of two conditions
- AND Both conditions need to be true
- OR Either condition may be true
- NOT Negates operation
- BETWEEN min AND max
  - e.g. True if value is >= min and <= max

# SELECT Statement – Examples

- We can use IN keyword to find data within groups

```
SELECT  Name, Phone, Department
  FROM  EMPLOYEE WHERE Department
   IN )‘Accounting’, ‘Finance’, ‘Marketing’(;
```

| **Name** | **Phone** | **Department** |
|---|---|---|
| Mary Jacobs | 285-8879 | Accounting |
| Kenji Numoto | 287-0098 | Marketing |
| Heather Jones | 287-9981 | Finance |
| Rosalie Jackson | 285-1273 | Accounting |
| Kim Sung | 287-3222 | Marketing |

# SELECT Statement – Examples

- Similarly NOT IN keyword acts against the previous query
  SELECT Name, Phone, Department
  FROM EMPLOYEE WHERE Department
  NOT IN )‘Accounting’, ‘Finance’, ‘Marketing’(;

| Name | Phone | Department |
|---|---|---|
| James Nester | | Info Systems |
| Richard Wu | 287-0123 | Info Systems |

# Ranges, Wildcards, and Nulls in WHERE Clause

- We can use BETWEEN keyword for ranges

```
SELECT   Name, Department
FROM     EMPLOYEE
WHERE EmployeeNumber BETWEEN 200 AND 500;
```

| Name | Department |
|---|---|
| Kenji Numoto | Marketing |
| Heather Jones | Finance |
| Rosalie Jackson | Accounting |
| James Nestor | Info Systems |

# Ranges, Wildcards, and Nulls in WHERE Clause

- This query is similar to the previous one with no BETWEEN keyword (takes longer space and more work)

```
SELECT  Name, Department
    FROM      EMPLOYE
    WHERE     EmployeeNumber  >= 200
    AND       EmployeeNumber <= 500;
```

# Ranges, Wildcards, and Nulls in WHERE Clause

- We can use LIKE keyword to show a part of a value in a field
  - We can use the Underscore (_) wild card as a character place holder

```
SELECT  *  FROM  PROJECT
WHERE Name LIKE ‘Q_ Portfolio Analysis’;
```

| ProjectID | Name | Department | MaxHours |
|---|---|---|---|
| 1000 | Q3 Portfolio Analysis | Finance | 75.0 |
| 1500 | Q4 Portfolio Analysis | Finance | 110.0 |

# Ranges, Wildcards, and Nulls in WHERE Clause

- We can use the ‘%’ wildcard for showing one or more characters
  - In Access we use ‘?’ for one and ‘%’ for more characters
  - To show all employees with phone number starting by 285

```
SELECT        *        FROM   EMPLOYEE
WHERE         Phone LIKE     ‘285-%’;
```

| EmployeeNumber | Name | Phone | Department |
|---|---|---|---|
| 100 | Mary Jacobs | 285-8879 | Accounting |
| 400 | Rosalie Jackson | 285-1273 | Accounting |

# Ranges, Wildcards, and Nulls in WHERE Clause

- e.g. To show with department ending by ing

SELECT * FROM EMPLOYEE
WHERE Department LIKE ‘%ing’;

| EmployeeNumber | Name | Phone | Department |
|---|---|---|---|
| 100 | Mary Jacobs | 285-8879 | Accounting |
| 200 | Kenji Numoto | 287-0098 | Marketing |
| 400 | Rosalie Jackson | 285-1273 | Accounting |
| 700 | Kim Sung | 287-3222 | Marketing |

# Ranges, Wildcards, and Nulls in WHERE Clause

- To find records with null values we can use the ‘IS NULL’ wildcard as:

```
SELECT      Name, Department        FROM EMPLOYEE
WHERE       Phone  IS NULL;
```

| Name | Department |
|---|---|
| James Nester | Info Systems |

# Structured Query Language (SQL) 22

By: M Shuaib Zarinkhail 2010

# Sorting the Results

- We can use the 'ORDER BY' keywords for sorting a query result as:

```
SELECT Name, Department
   FROM EMPLOYEE
   ORDER BY Department;
```

| Name | Department |
|---|---|
| Mary Jacobs | Accounting |
| Rosalie Jackson | Accounting |
| Heather Jones | Finance |
| James Nestor | Info Systems |
| Richard Wu | Info Systems |
| Kenji Numoto | Marketing |
| Kim Sung | Marketing |

# Sorting the Results

- By default, SQL sorts data as ascending
  - We can type the ‘ASC’ keyword after field name

- If needed, we can use the ‘DESC’ keyword and show results in descending order

# Sorting the Results

- The result of this query is the same the previous but descending department name

```
SELECT    Name, Department
FROM      EMPLOYEE
ORDER BY  Department
  DESC;
```

| Name | Department |
|---|---|
| Kenji Numoto | Marketing |
| Kim Sung | Marketing |
| Richard Wu | Info Systems |
| James Nestor | Info Systems |
| Heather Jones | Finance |
| Rosalie Jackson | Accounting |
| Mary Jacobs | Accounting |

# Sorting the Results

- A sort can be implement on more than one field in a query as:

```
SELECT Name,
  Department
FROM     EMPLOYEE
ORDER BY
  Department DESC,
  Name ASC;
```

| **Name** | **Department** |
|---|---|
| Kenji Numoto | Marketing |
| Kim Sung | Marketing |
| James Nestor | Info Systems |
| Richard Wu | Info Systems |
| Heather Jones | Finance |
| Mary Jacobs | Accounting |
| Rosalie Jackson | Accounting |

# Data Aggregation

- We can aggregate and abbreviate data as:
  - Count data
  - Collect data
  - Find minimum data value
  - Find maximum data value
  - Find the average of a data range
  - …

# Data Aggregation

To achieve the mentioned goals

- We can use the ‘Aggregate Functions’
- Aggregate Functions are ‘built-in’
  - They differ in DBMSs
  - MySQL has 5 built-in functions (next slide)
- Aggregate Functions use arithmetic operations on data and show the results
  - While using these functions, we should use the ‘Group By’ keywords in that query

# Data Aggregation

- We can aggregate and abbreviate data as:
  - Count data
  - Collect data
  - Find minimum data value
  - Find maximum data value
  - Find the average of a data range
  - …

# Data Aggregation

- MySQL has 5 built-in Aggregate Functions
  - COUNT (*[ALL | DISTINCT] expression)
  - SUM ([ALL | DISTINCT]expression)
  - AVG ([ALL | DISTINCT]expression)
  - MAX (expression)
  - MIN (expression)

# Data Aggregation

- This query counts employees in each department

```
SELECT    Department,      COUNT(*) FROM
     EMPLOYEE
           GROUP BY  Department;
```

| Department | Count(*) |
|---|---|
| Accounting | 2 |
| Marketing | 2 |
| Finance | 1 |
| Info Systems | 2 |

# Data Aggregation

- Shows the more than one employees regarding to the counted column

SELECT Department, COUNT(*) FROM EMPLOYEE

GROUP BY  Department HAVING   COUNT(*) > 1;

| Department | COUNT(*) |
|---|---|
| Accounting | 2 |
| Marketing | 2 |
| Info Systems | 2 |

# Data Aggregation

- COUNT() and SUM() functions are different
  - COUNT(): Counts the number of records
  - SUM(): Calculates the total of numeric fields values
  - *Example, Next Slide*

# Data Aggregation

```
SELECT COUNT(MaxHours) ‘All
  Records’, SUM(MaxHours) ‘Total
  Hours’
      FROM PROJECT;
```

| All Records | Total Hours |
|---|---|
| 4 | 468.00 |

# Data Aggregation

- We can use the DISTINCT keyword
  - Compare these two examples

1. SELECT COUNT(Department) FROM PROJECT;
2. SELECT COUNT(DISTINCT Department) FROM PROJECT;

| COUNT (Department) |
|---|
| 4 |

| COUNT (DISTINCT Department) |
|---|
| 3 |

# Data Aggregation

- MIN(), MAX(), & AVG() examples:

```
SELECT MIN(MaxHours) ‘Lowest Hours’,
  MAX(MaxHours), SUM(MaxHours)
  FROM PROJECT
  WHERE ProjectID < 1500;
```

| Lowest Hours | MAX(MaxHours) | SUM(MaxHours) |
|---|---|---|
| 75.00 | 145.00 | 358.00 |

# Data Aggregation

- We can not use aggregate functions after the WHERE clause in a query
  - e.g. This command is prohibited:

```
… WHERE MaxHours <
    AVG(MaxHours);
```

# Data Aggregation

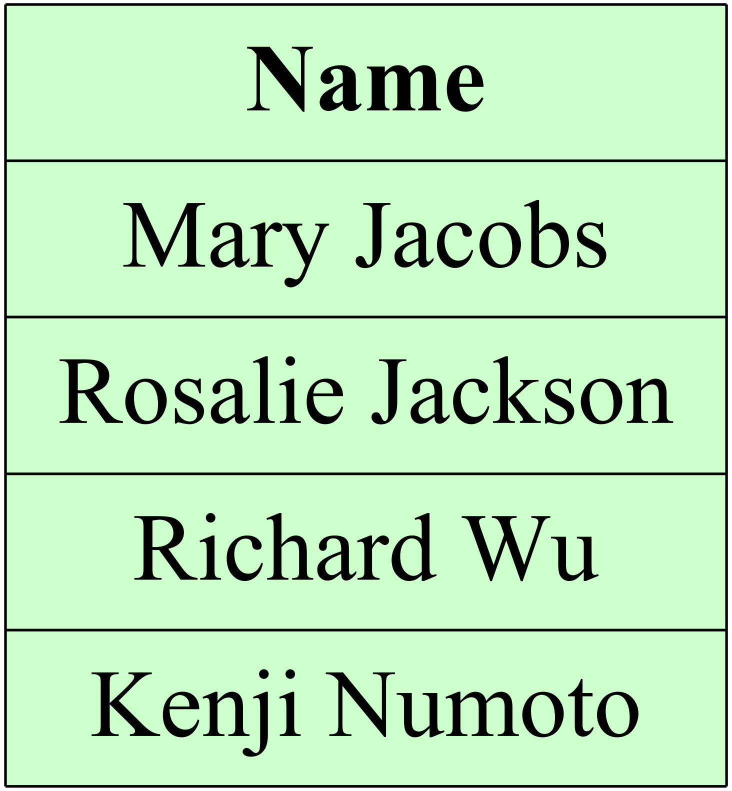
- Important points regarding the usage of Aggregate Functions
  - Assign column names while using aggregate functions (these function leave column names empty)
  - Be careful! Aggregate functions that calculate or average values, ignore NULLs in tables

# Subqueries

- We can use one or many tables in a single query
  - e.g. Show employee names, who had worked more than 40 hours on every assignment
  - To query this, data from two tables is needed
  - We can use subquery
    - *Example (Next Slide)*

# Subqueries

```
SELECT Name
FROM EMPLOYEE
WHERE EmployeeNumber
  IN
(SELECT DISTINCT
  EmployeeNum
  FROM ASSIGNMENT
  WHERE HoursWorked >
  40);
```



| Name |
|---|
| Mary Jacobs |
| Rosalie Jackson |
| Richard Wu |
| Kenji Numoto |

# Subqueries

- We can use the subquery method to design queries; hence, a simpler method is using joins instead of subqueries
  - e.g. This query joins two tables

```
SELECT Name, HoursWorked
   FROM EMPLOYEE, ASSIGNMENT
   WHERE
   EmployeeNumber =
   EmployeeNum;
```

| **Name** | **HoursWorked** |
|---|---|
| Mary Jacobs | 17.50 |
| Mary Jacobs | 45.75 |
| Kenji Numoto | 75.00 |
| Heather Jones | 12.50 |
| Rosalie Jackson | 8.00 |
| Rosalie Jackson | 70.50 |
| James Nestor | 20.25 |
| James Nestor | 25.25 |
| Richard Wu | 40.50 |
| Kim Sung | 20.25 |

# End of Database Two Course

Good Luck!