UVA | McIntire School of Commerce

SQL BASICS WITH THE SMALLBANKDB

STEFANO GRAZIOLI & MIKE MORRIS

This handout covers the most important SQL statements. The examples provided throughout are based on the SmallBank database.

Instructions on how to connect to the database will be distributed separately.

1. SINGLE TABLE QUERIES

- Tables and columns have short and full names. The full name is in the form schemaName.tableName.columnName. For example

  sg6m.LOAN.rate

  Depending on how the defaults are set, sometimes you can skip the schema name and/or the table name and just write

  rate

  If there is the possibility of confusion, a fuller name is required.

SELECTING DATA FROM A TABLE

1) Choosing all fields (columns)

```
SELECT *
FROM table_name;

SELECT *
FROM Customer;
```

- The ; at the end of the query is optional in most DBMS implementations.

2) Choosing a selected list of fields (columns)

```
SELECT column_name [, column_name, …]
FROM table_name;

SELECT f_name, l_name, date_of_birth
FROM Customer;
```

- The order in which you list the columns affects how they are presented in the resulting output.
- Items within [ ] are optional.

3) Temporarily renaming columns in query results

```
SELECT column_name AS 'column heading' [, column_name AS
'column_heading']
FROM table_name;
```

Example:

```
SELECT f_name as 'Customer Name'
FROM Customer;
```

4) Formatting results

```
SELECT format(date_due, 'd'), format(principal, 'C2')
FROM loan
ORDER BY date_due;
```

- Google 'transact-SQL format' to find more formatting commands

5) Including calculated columns in the results

```
SELECT date_due, rate, principal, rate * principal
FROM loan;
```

- If necessary, use parentheses to clarify the order of precedence in a computation, as in a * (b+c)

6) Eliminating duplicate query results with *DISTINCT*

If you use the keyword *distinct* after the keyword SELECT, you will only get unique rows. Example:

```
SELECT rate
FROM Loan;
```

VS.

```
SELECT DISTINCT rate
FROM Loan;
```

7) Selecting rows: the *WHERE* clause

```
SELECT Select_list
FROM table
WHERE search_conditions;
```

Example:

```
SELECT *
FROM Customer
WHERE f_name = 'Carl';
```

- In SQL, text is delimited by single quotes, as in 'Carl'

8) Selecting only the first n rows: TOP

```
SELECT TOP n Select_list
FROM table
WHERE search_conditions;
```

Example:

```
SELECT TOP 15 *
FROM Customer;
```

AVAILABLE SEARCH CONDITIONS OPERATORS

Comparison operators (=, <, >, !=. <>, <= ,>=)

```
SELECT * FROM loan
WHERE principal > 100000;
```

- Ranges (**between** and **not between**; inclusive of the end values)

```
SELECT * FROM loan
WHERE rate BETWEEN 7.5 AND 8.5;
```

- Lists (**in** and **not in**)

```
SELECT *
FROM Customer
WHERE city IN ('Charlottesville', 'Roanoke', 'Lexington');
```

- Character matches (**like** and **not like**)

```
SELECT f_name, l_name
```

```
FROM Customer
WHERE l_name LIKE 'Fos%';

SELECT f_name, l_name
FROM Customer
WHERE l_name LIKE '_oster';
```

- "**%**" (matches any string of zero or more characters) and "**_**" (matches any one character). In addition to those, brackets can be used to include either ranges or sets of characters.
- If you are not using the wildcards "**%**" or "**_**", you should use **=**, rather than **LIKE**. It is better and faster.
- Combinations of previous options using logical operators **and**, **or**, and **not** are possible:

```
SELECT f_name, l_name
FROM Customer
WHERE l_name LIKE 'Fos%' AND City NOT IN
('Charlottesville', 'Richmond');
```

SUMMARIZING, GROUPING, AND SORTING QUERY RESULTS

1) Aggregate functions

- Types of aggregate functions: sum, avg, count, count(*), max, min

```
SELECT SUM (principal) FROM loan;

SELECT AVG (rate) FROM loan;

SELECT MIN(rate), MAX(rate), COUNT(rate)
FROM loan;
```

- The **where** clause can be used to define the set of rows to which the aggregate functions apply

```
SELECT AVG (principal)
FROM loan
WHERE rate > 8.5;
```

- Difference between **count** and **count(*)**: **count** returns the number of non-null values in a specific column, whereas **count(*)** returns the number of rows.

```
SELECT COUNT(*) FROM customers;

SELECT COUNT(city) FROM customers;
```

- The keyword **distinct** can be used with sum, avg, and count to eliminate duplicate values before making calculations. Distinct appears inside the parenthesis and before the column name.

```
SELECT COUNT(DISTINCT city) FROM customers;
```

2) Using aggregate functions with groupings

- The **group by** clause can be used to organize a table into groups and get results separately for each group.
- **If you use aggregate functions in a SELECT query (e.g., SUM, AVG), only columns that appear in the 'Group By' can be visualized with the aggregations.**

```
SELECT rate, AVG(principal)
FROM loan
GROUP BY rate;
```

- The **where** clause can be used in a statement with **group by**. Only those rows that satisfy the condition will be included in the grouping.

```
SELECT rate, AVG(principal)
FROM loan
WHERE principal > 50000000
GROUP BY rate;
```

- The types of groups included in the answer set can be limited with the having keyword. ***Having*** *sets conditions for groups in the same way* ***where*** *sets conditions for individual rows*. Aggregate functions can be used in a **having** clause.

```
SELECT rate, AVG(principal)
FROM loan
GROUP BY rate
HAVING AVG(principal) > 50000000;
```

3) Sorting query results with the order by clause

- An **order by** clause is used to request the results of data retrieval in either ascending (**ASC**, which is the default) or descending (**DESC**) order by one or several columns

```
SELECT *
FROM loan
ORDER BY rate;
```

- Multiple sorts are possible

```
Select top 50 PERCENT lo.lo_id, lo.f_name, lo.l_name,
format (sum(rate/100*principal), 'C') as "Total interest"
```

```
from LOAN l, LOAN_OFFICER lo
where lo.lo_id = l.lo_id
group by lo.lo_id, lo.f_name, lo.l_name
order by sum(rate/100*principal) DESC
```

2. MULTIPLE TABLE QUERIES

SELECTING DATA FROM MULTIPLE TABLES: RELATIONAL JOINS

- Relational joins are SQL commands that combine data from multiple tables
- A "join" combinines the data in two tables by using the values in one column in the first table and matching them with the values of another column in the second table. In the most common case, a join matches a foreign key in one table and the primary key in another.
- Queries that include multiple joins are possible. These queries "hop" from one table to the next, to the next, to the next.

1) Joining tables using a foreign key/primary key combination

```
SELECT l_id, principal, date_due, loan_officer.lo_id,
l_name
FROM loan, loan_officer
WHERE loan.lo_id = loan_officer.lo_id;
```

- Table name qualifiers (loan and loan_officer in the example above) are used when a column name is not unique and we must clarify which column we are referring to. The format is *tableName.attributeName*
- The where clause restricts the entries to those where the join condition is true.
- If the **where** clause is (accidentally) omitted, SQL returns a result that contains the "Cartesian product" of the tables, i.e., all possible combinations of all the rows from all the tables. Thus, if the loan_officer table contained 30 entries and the loan table contained 100 entries, the Cartesian product consists of (30x100=) 3,000 entries. This is very rarely what you intended. Bottom line: remember to include the **where** clause!
- The column set to be displayed can come from either one of the tables, or from both.
- There are several styles to write joins. You might be familiar with a different one. That is ok. Feel free to use the one that you prefer.

2) Alternative syntax

```
SELECT l_id, principal, date_due, loan_officer.lo_id,
l_name
FROM loan
INNER JOIN loan_officer ON
loan.lo_id = loan_officer.lo_id;
```

- INNER JOIN produce the same resutl as the previous query.

3) All records from the 'left' Table, even if some do not match the other Table

```
SELECT l_id, principal, date_due, loan_officer.lo_id,
l_name
FROM loan
LEFT JOIN loan_officer ON
loan.lo_id = loan_officer.lo_id;
```

- Which table is the 'left' table is arbitrary. You pick it. It is the one that we think of as 'anchoring' the join. The one we start from.
- LEFT JOIN includes in the results the records from the left table **even if they have no match** in the other table. LEFT OUTER JOIN produce the same result.

4) All records from both Tables, even if some do not match the other table

```
SELECT l_id, principal, date_due, loan_officer.lo_id,
l_name
FROM loan
FULL JOIN loan_officer ON
loan.lo_id = loan_officer.lo_id;
```

- FULL JOIN includes in the results the records from the both tables **even if they have no match** in the other table.

5) Adding elements to the *where* clause

```
SELECT l_id, principal, date_due, loan_officer.lo_id,
l_name
FROM loan, loan_officer
WHERE loan.lo_id = loan_officer.lo_id
AND principal > 10000000;
```

```
SELECT l_id, principal, date_due, loan_officer.lo_id,
l_name
FROM loan
LEFT JOIN loan_officer ON
loan.lo_id = loan_officer.lo_id
WHERE principal > 10000000;
```

- Any combination of logical operators can be used to add conditions in the **where** clause

6) Joining three or more tables

- Joins are not limited to two tables; however, you will seldom see queries with more than 6 or 7 tables joined together. "Normal" is 2-4 tables. Here is an example with 4 tables.

```
SELECT customer.f_name, customer.l_name
FROM loan_officer, loan, customer_in_loan, customer
WHERE loan_officer.l_name = 'Romani'
AND loan_officer.lo_id = loan.lo_id
AND loan.l_id = customer_in_loan.l_id
AND customer_in_loan.c_id = customer.c_id;
```

- The columns used to join the tables (order number and product number above) may be included in the ***select*** statement but do not have to be.
- What does this query compute? Make sure that you understand.

NESTED QUERIES

- It is possible to feed the result of a query directly into another query.

```
SELECT *
FROM loan
WHERE principal > ( SELECT avg(principal)
                    FROM LOAN );
```

UNION

- It is possible to append the results of two queries if the columns are compatible.
- To simplify, Joins add columns to a table, while Unions add rows to a table.

```
SELECT l.l_id, l.principal
FROM LOAN l
WHERE principal < 50000
UNION
SELECT l.l_id, l.principal
```

```
FROM LOAN l
WHERE principal > 300000
```

- This query may have been written using WHERE .... OR. It is written this way to demonstrate how to use UNION.
- The queries that are appended through UNON can contain Order By only once, at the end

3. INSERTING, UPDATING and DELETING rows

INSERTING A NEW ROW INTO A TABLE

```
INSERT INTO table_name (column1, column2, column3…)
VALUES (value1, value2, value3, …)
```

- If the order of the values is the same as the order of the columns in the table, the column specification can be omitted (see example below)
- Strings are delimited by single quotes
- You may need permission from the sys admin to insert rows in a table.

```
INSERT INTO Customer
VALUES (2323, 'John', 'Smith', 'Cville', 'VA')
```

Or

```
INSERT INTO Customer (f_name, l_name, c_id, state, city)
VALUES ('John', 'Smith', 2323, 'VA', 'Cville')
```

- Inserting a duplicated primary key will give you an error.

UPDATING ONE OR MORE ROW IN A TABLE

```
UPDATE table_name
SET column = value
WHERE condition
```

- Depending on the condition, one or more (or none) rows will be changed
- Careful with UPDATE! There is no 'undo'

```
UPDATE Customer
SET f_name = 'Jane'
WHERE c_id = 2323
```

```
UPDATE Customer
SET city = 'Charlottesville'
WHERE city = 'Cville'
```

DELETING ONE OR MORE ROW IN A TABLE

```
DELETE FROM table_name
WHERE condition
```

- Depending on the condition, one or more (or none) rows will be changed
- **Careful with DELETE!** There is no 'undo' and if you forget to specify the condition, the whole table will be cleared.

```
DELETE FROM Customer
WHERE c_id = 2323
```

- The next example deletes multiple rows. If you forget to specify the condition, the whole table will be cleared. Again, there is no 'undo'.

```
DELETE FROM Customer
WHERE city = 'Cville'
```