SQL OVERVIEW

CS121: Introduction to Relational Database Systems

Fall 2014 – Lecture 4

SQL

- SQL = Structured Query Language
- Original language was "SEQUEL"
  - IBM's System R project (early 1970's)
  - "Structured English Query Language"
- Caught on very rapidly
  - Simple, declarative language for writing queries
  - Also includes many other features
- Standardized by ANSI/ISO
  - SQL-86, SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2008, SQL:2011
  - Most implementations *loosely* follow the standards (plenty of portability issues)

SQL Features

- Data Definition Language (DDL)
  - Specify relation schemas (attributes, domains)
  - Specify a variety of integrity constraints
  - Access constraints on data
  - Indexes and other storage "hints" for performance
- Data Manipulation Language (DML)
  - Generally based on relational algebra
  - Supports querying, inserting, updating, deleting data
  - Very sophisticated features for multi-table queries
- Other useful tools
  - Defining views, transactions, etc.

SQL Basics

- SQL language is case-insensitive
  - both keywords and identifiers (for the most part)
- SQL statements end with a semicolon
- SQL comments have two forms:
  - Single-line comments start with two dashes

    ```
    -- This is a SQL comment.
    ```
  - Block comments follow C style

    ```
    /*
     * This is a block comment in SQL.
     */
    ```

SQL Databases

- SQL relations are contained within a database
  - Each application usually works against its own database
  - Several applications may share the same database, too
- An example from MySQL:

```
CREATE DATABASE bank;
USE bank;
```

  - Creates a new, empty database called **bank**
  - **USE** statement makes **bank** the “default” database for the current connection
  - DDL and DML operations will be evaluated in the context of the connection’s default database

Creating a SQL Table

- In SQL, relations are called “tables”
  - Not *exactly* like relational model “relations” anyway
- Syntax:

```
CREATE TABLE t (
    attr1 domain1,
    attr2 domain2,
    ... ,
    attrN domainN
);
```

  - **`t`** is name of relation (table)
  - **`attr1`**, … are names of attributes (columns)
  - **`domain1`**, … are domains (types) of attributes

SQL Names

- Tables, columns, etc. require names
- Rules on valid names can vary dramatically across implementations

- Good, portable rules:
  - First character should be alphabetical
  - Remaining characters should be alphanumeric or underscore '_'
  - Use same the case in DML that you use in DDL

SQL Attribute Domains

- Some standard SQL domain types:

  `CHAR(N)`
  - A character field, fixed at N characters wide
  - Short for **`CHARACTER(N)`**

  `VARCHAR(N)`
  - A variable-width character field, with maximum length N
  - Short for **`CHARACTER VARYING(N)`**

  `INT`
  - A signed integer field (typically 32 bits)
  - Short for **`INTEGER`**
  - Also **`TINYINT`**, **`SMALLINT`**, **`BIGINT`**, etc.
  - Also unsigned variants
    - Non-standard, only supported by some vendors

CHAR vs. VARCHAR

- Both **CHAR** and **VARCHAR** have a size limit
- **CHAR** is a fixed-length character field
  - Can store shorter strings, but storage layer pads out the value to the full size
- **VARCHAR** is a variable-length character field
  - Storage layer doesn't pad out shorter strings
  - String's length must also be stored for each value
- Use **CHAR** when all values are approximately (or *exactly*) the same length
- Use **VARCHAR** when values can be very different lengths

SQL Attribute Domains (2)

- More standard SQL domain types:

  `NUMERIC(P,D)`
  - A fixed-point number with user-specified precision
  - P total digits; D digits to right of decimal place
  - Can exactly store numbers

  `DOUBLE PRECISION`
  - A double-precision floating-point value
  - An approximation! Don't use for money! ☺
  - **`REAL`** is sometimes a synonym

  `FLOAT(N)`
  - A floating-point value with at least N bits of precision

SQL Attribute Domains (3)

- Other useful attribute domains, too:

  `DATE`, `TIME`, `TIMESTAMP`
  - For storing temporal data
- Large binary/text data fields

  `BLOB`, `CLOB`, `TEXT`
  - Binary Large Objects, Character Large Objects
  - Large text fields
  - **`CHAR`, `VARCHAR`** tend to be very limited in size
- Other specialized types
  - Enumerations, geometric or spatial data types, etc.
  - User-defined data types

Choosing the Right Type

- Need to think carefully about what type makes most sense for your data values
- Example: storing ZIP codes
  - US postal codes for mail routing
  - 5 digits, e.g. 91125 for Caltech
- Does `INTEGER` make sense?
- **Problem 1:** Some ZIP codes have leading zeroes!
  - Many east-coast ZIP codes start with 0.
  - Numeric types won't include leading zeros.
- **Problem 2:** US mail also uses ZIP+4 expanded ZIP codes
  - e.g. 91125-8000
- **Problem 3:** Many foreign countries use non-numeric values

Choosing the Right Type (2)

- Better choice for ZIP codes?
  - A **CHAR** or **VARCHAR** column makes much more sense
- For example:
  - **CHAR(5)** or **CHAR(9)** for US-only postal codes
  - **VARCHAR(20)** for US + international postal codes
- Another example: monetary amounts
  - Floating-point representations cannot exactly represent all values
    - e.g. 0.1 is an infinitely-repeating binary decimal value
  - Use **NUMERIC** to represent monetary values

Example SQL Schema

- Creating the account relation:

```
CREATE TABLE account (
    acct_id       CHAR(10),
    branch_name   CHAR(20),
    balance       NUMERIC(12, 2)
);
```

  - Account IDs can't be more than 10 chars
  - Branch names can't be more than 20 chars
  - Balances can have 10 digits left of decimal, 2 digits right of decimal
    - Fixed-point, exact precision representation of balances

Inserting Rows

- Tables are initially empty
- Use **INSERT** statement to add rows

```
INSERT INTO account
   VALUES ('A-301', 'New York', 350);
INSERT INTO account
   VALUES ('A-307', 'Seattle', 275);
...
```

  - String values are <u>single-quoted</u>
  - (In SQL, double-quoted strings refer to column names)
  - Values appear in same order as table's attributes

Inserting Rows (2)

- Can specify which attributes in **INSERT**

```
INSERT INTO account (acct_id, branch_name, balance)
  VALUES ('A-301', 'New York', 350);
```

  - Can list attributes in a different order
  - Can exclude attributes that have a default value
- Problem: We can add multiple accounts with same account ID!

```
INSERT INTO account
  VALUES ('A-350', 'Seattle', 800);
INSERT INTO account
  VALUES ('A-350', 'Los Angeles', 195);
```

Primary Key Constraints

- The **CREATE TABLE** syntax also allows integrity constraints to be specified
  - Are often specified after all attributes are listed
- Primary key constraint:

```
CREATE TABLE account (
    acct_id        CHAR(10),
    branch_name    CHAR(20),
    balance        NUMERIC(12, 2),

    PRIMARY KEY (acct_id)
);
```

  - Database won't allow two rows with same account ID

Primary Key Constraints (2)

- A primary key can have multiple attributes

```
CREATE TABLE depositor (
  customer_name  VARCHAR(30),
  acct_id        CHAR(10),
  PRIMARY KEY (customer_name, acct_id)
);
```

  - Necessary because SQL tables are multisets
- A table cannot have multiple primary keys
  - (obvious)
- *Many* other kinds of constraints too
  - Will cover in future lectures!

Removing Rows, Tables, etc.

- Can delete rows with **`DELETE`** command
  - Delete bank account with ID A-307:

```
DELETE FROM account WHERE acct_id = 'A-307';
```

  - Delete all bank accounts:

```
DELETE FROM account;
```

- Can drop tables and databases:
  - Remove account table:

```
DROP TABLE account;
```

  - Remove an entire database, including all tables!

```
DROP DATABASE bank;
```

Issuing SQL Queries

- SQL queries use the **SELECT** statement
- *Very* central part of SQL language
  - Concepts appear in all DML commands
- General form is:

```
SELECT A1, A2, ...
  FROM r1, r2, ...
  WHERE P;
```

  - r_i are the relations (tables)
  - A_i are attributes (columns)
  - P is the selection predicate

SELECT Operations

- **SELECT** $A_1, A_2, \ldots$
  - Corresponds to a relational algebra project operation

    $\Pi_{A_1,\, A_2,\, \ldots}(\ \ldots\ )$
  - Some books call σ "restrict" because of this name mismatch
- **FROM** $r_1, r_2, \ldots$
  - Corresponds to Cartesian product of relations $r_1, r_2, \ldots$

    $r_1 \times r_2 \times \ldots$

`SELECT` Operations (2)

- **`WHERE P`**
  - Corresponds to a selection operation
    $\sigma_P(\ \ldots\ )$
  - Can be omitted. When left off, P = true
- Assembling it all:
  ```
  SELECT A1, A2, ... FROM r1, r2, ...
    WHERE P;
  ```
  - Equivalent to: $\Pi_{A_1,\, A_2,\, \ldots}(\sigma_P(r_1 \times r_2 \times \ldots))$

SQL and Duplicates

- Biggest difference between relational algebra and SQL is use of multisets
  - In SQL, relations are <u>multisets</u> of tuples, not sets
- Biggest reason is practical:
  - Removing duplicate tuples is time consuming!
- Must revise definitions of relational algebra operations to handle duplicates
  - Mainly affects set-operations: $\cup$, $\cap$, $-$
  - (Book explores this topic in depth)
- SQL provides ways to exclude duplicates for all operations

Example Queries

"Find all branches with at least one bank account."

```
SELECT branch_name
  FROM account;
```

```
+-------------+
| branch_name |
+-------------+
| New York    |
| Seattle     |
| Los Angeles |
| New York    |
| Los Angeles |
+-------------+
```

- Equivalent to typing:

```
SELECT ALL branch_name
  FROM account;
```

- To eliminate duplicates:

```
SELECT DISTINCT branch_name
  FROM account;
```

```
+-------------+
| branch_name |
+-------------+
| New York    |
| Seattle     |
| Los Angeles |
+-------------+
```

Selecting Specific Attributes

- Can specify one or more attributes to appear in result

  "Find ID and balance of
  all bank accounts."

  ```
  SELECT acct_id, balance
    FROM account;
  ```

acct_id	balance
A-301	350.00
A-307	275.00
A-318	550.00
A-319	80.00
A-322	275.00

- Can also specify * to mean "all attributes"

  ```
  SELECT * FROM account;
  ```

  - Returns all details of all accounts.

acct_id	branch_name	balance
A-301	New York	350.00
A-307	Seattle	275.00
A-318	Los Angeles	550.00
A-319	New York	80.00
A-322	Los Angeles	275.00

Computing Results

- The **SELECT** clause is a *generalized projection* operation
  - Can compute results based on attributes

```
SELECT cred_id, credit_limit – balance
  FROM credit_account;
```

  - Computed values don't have a (standard) name!
    - Many DBMSes name the 2nd column "**credit_limit – balance**"
- Can also name (or rename) values

```
SELECT cred_id,
       credit_limit – balance AS available_credit
  FROM credit_account;
```

WHERE Clause

- The **WHERE** clause specifies a selection predicate
  - Can use comparison operators:

=, <>	equals, not-equals (!= also usually supported)
<, <=	less than, less or equal
>, >=	greater than, greater or equal

  - Can refer to any attribute in **FROM** clause
  - Can include arithmetic expressions in comparisons

WHERE Examples

"Find IDs and balances of all accounts in the Los Angeles branch."

```
SELECT acct_id, balance FROM account
  WHERE branch_name = 'Los Angeles';
```

```
+---------+---------+
| acct_id | balance |
+---------+---------+
| A-318   |  550.00 |
| A-322   |  275.00 |
+---------+---------+
```

"Retrieve all details of bank accounts with a balance less than $300."

```
SELECT * FROM account
  WHERE balance < 300;
```

```
+---------+-------------+---------+
| acct_id | branch_name | balance |
+---------+-------------+---------+
| A-307   | Seattle     |  275.00 |
| A-319   | New York    |   80.00 |
| A-322   | Los Angeles |  275.00 |
+---------+-------------+---------+
```

Larger Predicates

- Can use **AND**, **OR**, **NOT** in **WHERE** clause

```
SELECT acct_id, balance FROM account
  WHERE branch_name = 'Los Angeles' AND
        balance < 300;
```

```
SELECT * FROM account
  WHERE balance >= 250 AND balance <= 400;
```

- SQL also has **BETWEEN** and **NOT BETWEEN** syntax

```
SELECT * FROM account
  WHERE balance BETWEEN 250 AND 400;
```

  - Note that **BETWEEN** <u>includes</u> interval endpoints!

String Comparisons

- String values can be compared
  - Lexicographic comparisons
  - Default is often to <u>ignore</u> case!

```
SELECT 'HELLO' = 'hello';  -- Evaluates to true
```

- Can also do pattern matching with **`LIKE`** expression

  `string_attr` **`LIKE`** ***`pattern`***

  - **`pattern`** is a string literal enclosed in single-quotes
    - `%` (percent) matches a substring
    - `_` (underscore) matches a single character
    - Can escape `%` or `_` with a backslash \

String-Matching Example

"Find all accounts at branches with 'le' somewhere in the name."

- Why? I don't know…

```
SELECT * FROM account
  WHERE branch_name LIKE '%le%';
```

```
+---------+-------------+---------+
| acct_id | branch_name | balance |
+---------+-------------+---------+
| A-307   | Seattle     |  275.00 |
| A-318   | Los Angeles |  550.00 |
| A-322   | Los Angeles |  275.00 |
+---------+-------------+---------+
```

String Operations

- Regular-expression matching is also part of the SQL standard (SQL:1999)
- String-matching operations tend to be expensive
  - Especially patterns with a leading wildcard, e.g. `'%abc'`
- Try to avoid heavy reliance on pattern-matching

- If string searching is required, try to pre-digest text and generate search indexes
  - Some databases provide "full-text search" capabilities, but such features are vendor-specific!

FROM Clause

- Can specify one or more tables in **FROM** clause
- If multiple tables:
  - Select/project against Cartesian product of relations

```
-- Produces a row for every combination
-- of input tuples.
SELECT * FROM borrower, loan;
```

```
+-----------+---------+---------+---------------+---------+
| cust_name | loan_id | loan_id | branch_name   | amount  |
+-----------+---------+---------+---------------+---------+
| Anderson  | L-437   | L-419   | Seattle       | 2900.00 |
| Jackson   | L-419   | L-419   | Seattle       | 2900.00 |
| Lewis     | L-421   | L-419   | Seattle       | 2900.00 |
| Smith     | L-445   | L-419   | Seattle       | 2900.00 |
| Anderson  | L-437   | L-421   | San Francisco | 7500.00 |
| Jackson   | L-419   | L-421   | San Francisco | 7500.00 |
| Lewis     | L-421   | L-421   | San Francisco | 7500.00 |
| ...                                                     |
```

FROM Clause (2)

- If tables have overlapping attributes, use **`tbl_name.attr_name`** to distinguish

```
SELECT * FROM borrower, loan
  WHERE borrower.loan_id = loan.loan_id;
```

```
+-----------+---------+---------+---------------+---------+
| cust_name | loan_id | loan_id | branch_name   | amount  |
+-----------+---------+---------+---------------+---------+
| Jackson   | L-419   | L-419   | Seattle       | 2900.00 |
| Lewis     | L-421   | L-421   | San Francisco | 7500.00 |
| Anderson  | L-437   | L-437   | Las Vegas     | 4300.00 |
| Smith     | L-445   | L-445   | Los Angeles   | 2000.00 |
+-----------+---------+---------+---------------+---------+
```

  - All columns can be referred to by **`tbl_name.attr_name`**
- This kind of query is called an <u>equijoin</u>
- Databases optimize equijoin queries *very* effectively.

SQL and Joins

- SQL provides several different options for performing joins across multiple tables
- This form is the most basic usage
  - Was in earliest versions of SQL
  - Doesn't provide natural joins
  - Can't do outer joins either
- Will cover other forms of SQL join syntax soon…

Renaming Tables

- Can specify alternate names in **FROM** clause too
  - Write: **`table AS name`**
  - (The **AS** is optional, but it's clearer to leave it in.)
- Previous example:

  "Find the loan with the largest amount."
  - Started by finding loans that have an amount smaller than some other loan's amount
  - Used Cartesian product and rename operation

```
SELECT DISTINCT loan.loan_id
  FROM loan, loan AS test
  WHERE loan.amount < test.amount;
```

```
+---------+
| loan_id |
+---------+
| L-445   |
| L-419   |
| L-437   |
+---------+
```

Renaming Tables (2)

- When a table is renamed in **FROM** clause, can use the new name in both **SELECT** and **WHERE** clauses
- Useful for long table names! ☺

```
SELECT c.cust_name, l.amount
  FROM customer AS c, borrower AS b,
       loan AS l
  WHERE c.cust_name = b.cust_name AND
        b.loan_id = l.loan_id;
```

Set Operations

- SQL also provides set operations, like relational algebra
- Operations take two relations and produce an output relation
- Set-union:

  select$_1$ **UNION** ***select***$_2$ ***;***
- Set-intersection:

  select$_1$ **INTERSECT** ***select***$_2$ ***;***
- Set-difference:

  select$_1$ **EXCEPT** ***select***$_2$ ***;***
- **Note:** ***select***$_i$ are complete **SELECT** statements!

Set-Operation Examples

- Find customers with an account or a loan:

```
SELECT cust_name FROM depositor UNION
SELECT cust_name FROM borrower;
```

  - Database automatically eliminates duplicates

- Find customers with an account but not a loan:

```
SELECT cust_name FROM depositor EXCEPT
SELECT cust_name FROM borrower;
```

  - Can also put parentheses around **SELECT** clauses for readability

```
 (SELECT cust_name FROM depositor)
EXCEPT
 (SELECT cust_name FROM borrower);
```

Set Operations and Duplicates

- By default, SQL set-operations <u>eliminate</u> duplicate tuples
  - Opposite to default behavior of **`SELECT`**!
- Can keep duplicate tuples by appending **`ALL`** to set operation:

  select$_1$ **`UNION ALL`** ***select***$_2$ **`;`**

  select$_1$ **`INTERSECT ALL`** ***select***$_2$ **`;`**

  select$_1$ **`EXCEPT ALL`** ***select***$_2$ **`;`**

How Many Duplicates?

- Need to define behavior of "set operations" on multisets
- Given two <u>multiset</u> relations r_1 and r_2
  - r_1 and r_2 have same schema
  - Some tuple t appears c_1 times in r_1, and c_2 times in r_2

$r_1 \cup_{ALL} r_2$

contains $c_1 + c_2$ copies of t

$r_1 \cap_{ALL} r_2$

contains $min(c_1, c_2)$ copies of t

$r_1 -_{ALL} r_2$

contains $max(c_1 - c_2, 0)$ copies of t

Other Relational Operations

- Can actually update definitions of all relational operations to support multisets
- Necessary for using relational algebra to model execution plans
- Not terribly interesting though… ☺

- If you're curious, see book for details

SQL Style Guidelines

- Follow good coding style in SQL!
- Some recommendations:
  - Use lowercase names for tables, columns, etc.
  - Put a descriptive comment above every table
  - Write all SQL keywords in uppercase
  - Follow standard indentation scheme
    - e.g. indent columns in table declarations by 2-4 spaces
  - Keep lines to 80 characters or less!
    - wrap lines in reasonable places

- **Note:** You <u>will</u> lose points for sloppy SQL.

Next Time

- Sorting results
- Grouping and aggregate functions
- Nested queries and many more set operations
- How to update SQL databases