CTEC323 Lecture 10

Dan Zingaro
OISE/UT

November 13, 2008

What is SQL?

- SQL is composed of commands that enable users to
  - create databases and tables (part of SQL's Data Definition Language; DDL)
  - perform data manipulation and administration, such as adding, deleting and modifying data (part of SQL's Data Manipulation Language; DML)
  - query the database to extract information (part of DML)
- We use SQL without knowing physical data storage format of database

Running Example

We will use two tables, `vendor` and `product`, that implement the following business rules

- Each vendor may supply many products (including no products)
- Each product is supplied by at most one vendor
- That is, product is optional to vendor, and vendor is optional to product

SQLite

- The RDBMS we will use in this class is called SQLite
- It is open-source, available for many platforms, and consists of a single executable file
- You simply run the executable file to start interacting with databases
- SQLite databases consist of just one file you can save on a flash drive or send by email
- Other databases like MySQL
  - Have more complicated installs and configurations
  - Run a server process in the background
  - Require you to login with a username and password
  - Make collaboration on databases more difficult because databases are not stored as single files

SQLite...

- To create a SQLite database, simply call the SQLite executable with the name of the database to use
- If it exists, SQLite will open it; otherwise, it will create it
- Example: `sqlite3 vendprod`
- You can type SQL statements at the SQLite prompt, but they will be lost when you close SQLite
- Instead, you can store SQL in a file and execute it with `.read filename`

Column Types

- We can specify a “type affinity” for each column of data, telling SQLite what type of data we intend to store there
  - `text`
  - `numeric`
  - `integer`
  - `real`
- If we specify no type affinity, all types of data can be stored

Creating Tables: vendprod.sql

```
CREATE TABLE VENDOR (
  V_CODE INTEGER PRIMARY KEY NOT NULL UNIQUE,
  V_NAME TEXT NOT NULL,
  V_AREACODE TEXT NOT NULL default ’416’,
  V_PHONE TEXT NOT NULL,
  UNIQUE (V_AREACODE, V_PHONE));

CREATE TABLE PRODUCT (
  P_CODE TEXT NOT NULL UNIQUE,
  P_DESCRIPT TEXT NOT NULL CHECK (length(P_DESCRIPT) >= 5),
  P_INDATE TEXT NOT NULL,
  P_QUANTITY INTEGER NOT NULL,
  V_CODE INTEGER,
  PRIMARY KEY (P_CODE),
  FOREIGN KEY (V_CODE) REFERENCES VENDOR (V_CODE));
```

Creating Tables...

- The `PRIMARY KEY` constraint specifies that a column or columns is the PK
- If the PK is one column, we can write it with the column constraint; otherwise, we write it after all column definitions
- The `NOT NULL` constraint means that we must provide a value for the column in each row
- The `DEFAULT` constraint specifies the value of the column if we do not give one when inserting a row
- The `UNIQUE` column constraint means that each value can appear at most once in the column
- The `UNIQUE` constraint can also be used to ensure that a combination of column values is unique
- `CHECK` constraints can validate data when an attribute value is entered

Entity and Referential Integrity

- SQLite enforces entity integrity if we include the `NOT NULL` designation on our PK fields
- Unlike other RDBMS's, SQLite does not enforce referential integrity (i.e. we can have a product referring to a nonexistent vendor)
- Some other RDBMS's support `ON DELETE CASCADE`, which means that if we delete a row from the parent table, the rows it refers to in the child table will also be deleted
- They may also support an `ON UPDATE CASCADE` feature: if we change a parent row's PK, matching child rows will have their FK's changed to reflect this

Indexes

- We can use indexes to (1) improve efficiency or (2) avoid duplicate column values
- When we used the UNIQUE constraint when creating a table, we were actually creating a unique index on the specified column(s)
- We can also add indexes once the table has been created, using CREATE INDEX. For example, if we did not add the unique constraint on V_AREACODE and V_PHONE when we created the table, we could add it as follows

```
CREATE UNIQUE INDEX PHONEINDEX ON VENDOR (V_AREACODE, V_PHONE);
```

- When we create an index on a column, searching in the table for rows matching criteria on that column will be very fast
- We can remove an index using DROP INDEX followed by the index name

Inserting into a Table: ddl-insert.sql

```
INSERT INTO VENDOR VALUES (1, ’Bryson, Inc.’, ’905’, ’1234567’);
INSERT INTO VENDOR (V_CODE, V_NAME, V_PHONE) VALUES (2, ’Smithson, Inc.’, ’7654321’);
INSERT INTO VENDOR (V_CODE, V_NAME, V_PHONE) VALUES (3, ’Danson, Inc.’, ’7257257’);

INSERT INTO PRODUCT (P_CODE, P_DESCRIPT, P_INDATE, P_QUANTITY)
  VALUES (1, ’water bottle’, ’2008-10-20’, 40);
INSERT INTO PRODUCT (P_CODE, P_DESCRIPT, P_INDATE, P_QUANTITY)
  VALUES (2, ’baseball glove’, ’2007-03-14’, 9);
INSERT INTO PRODUCT VALUES (3, ’laptop’, ’2008-10-22’, 3, 1);
```

- The first insert statement does not specify a list of attributes before `VALUES`, so we must enter data for each attribute
- The second and third insert statements show that we can specify only a subset of attributes
- We can only leave an attribute out when it has a `DEFAULT` value or it doesn’t have `NOT NULL`

Selecting from a Table: dml-select.sql

```
SELECT * FROM VENDOR;
SELECT P_DESCRIPT, P_INDATE FROM PRODUCT;
```

- The * means “all columns”
- Use `.headers on` so SQLite shows the names of columns

Updating a Table: dml-update.sql

```
UPDATE VENDOR
  SET V_PHONE = ’222-3333’
  WHERE V_CODE = 1;

UPDATE VENDOR
  SET V_AREACODE = 818, V_PHONE = ’111-4444’
  WHERE V_CODE = 2;
```

- We can specify multiple columns to change after `SET`
- `WHERE` tells us which rows to update
- Question: what if we leave out the `WHERE` clause?

Deleting from a Table: dml-delete.sql

```
DELETE from VENDOR WHERE V_CODE = 2;
DELETE FROM PRODUCT WHERE P_INDATE <= DATE (’2008-10-20’);
```

- The `DATE` function converts a string to a date
- Question: what if we leave out the `WHERE` clause?

SELECT Again: dml-select2.sql

```
SELECT * FROM PRODUCT WHERE P_CODE >= 1;
SELECT * FROM PRODUCT WHERE P_CODE IN (1, 2);
SELECT * FROM PRODUCT WHERE P_CODE >= 2 AND V_CODE = 1;
SELECT * FROM PRODUCT WHERE P_DESCRIPT <= ’c’;
SELECT * FROM PRODUCT WHERE P_INDATE <> DATE (’2008-10-20’);
SELECT P_DESCRIPT, P_QUANTITY * 2 AS DOUBLE_QUANTITY FROM PRODUCT;
```

- Select enables you to transform data into information
- `WHERE` clause lets you specify criteria on which to include rows
- Comparison operators: `=, <, <=, >, >=, <>`
- These operators can be used on character data too, comparing them alphabetically
- For example `’a’ < ’b’` and `’44’ < ’5’`
- We can use logical operators `AND, OR, NOT`
- We can use `IN` and `BETWEEN` to require that a value exist among the given possibilities
- It is possible to use “computed columns” and aliases (see the last example)
- Question: how can we rewrite the second query without using `IN`?

Pattern Matching on Strings: dml-like.sql

- LIKE allows you to use wildcards to find patterns in text attributes
- The % symbol is a placeholder for “any number of characters”
- The _ symbol is a placeholder that means “exactly one character”
- Examples
  - J% matches strings like JULY, JUNE BUG, J234
  - J%N matches strings like JASON, JAN, JN
  - J%N_ matches strings like JASONA, JANE, JNQ (but not JASON)
  - R__L matches RAIL, REEL, REAL

```
SELECT * FROM PRODUCT WHERE P_DESCRIPT LIKE '%a%l %';
```

Altering a Table: dml-alter.sql

```
ALTER TABLE PRODUCT ADD COLUMN UPC TEXT;
```

- We can add a column to a table using `ALTER TABLE`
- Other RDBMS's allow you to use `ALTER TABLE` to remove columns, change datatypes, and add or remove constraints
- We can remove a table with `DROP TABLE` followed by the `TABLENAME`

Ordering and Aggregating: dml-select3.sql

```
SELECT * FROM PRODUCT ORDER BY P_DESCRIPT;
SELECT COUNT(V_CODE) FROM PRODUCT;
SELECT MAX(P_QUANTITY) FROM PRODUCT WHERE P_CODE >= 2;
```

- We can order the results of a select by using `ORDER BY`
- If present, we write `ORDER BY` just prior to the optional `LIMIT` and `OFFSET` clauses in the `SELECT` statement
- If the column we order on has duplicates, we can further order the rows by including more comma-separated columns
- `COUNT(*)` counts the rows in a query result set
- `COUNT(COLUMN)` counts the number of non-NULL values in a given column
- Also available: `MAX(COLUMN)`, `MIN(COLUMN)`, `SUM(COLUMN)`, `AVG(COLUMN)`

Subqueries: dml-subquery.sql

- A subquery is a query nested in another query
- The inner query is always executed first, and its output is the input to the outer query
- The following query lists the V_CODE and V_NAME of only those vendors that supply at least one product

```
SELECT V_CODE, V_NAME FROM VENDOR
  WHERE V_CODE IN (SELECT V_CODE FROM PRODUCT);
```

- The inner query creates a table consisting of the V_CODE values found in PRODUCT
- The outer query collects all rows from VENDOR whose V_CODE exists in that inner query (i.e. exists in the product table)

Subqueries...: dml-subquery2.sql

- The following query gives the names of products with maximum QUANTITY

```
SELECT P_CODE, P_DESCRIPT, P_QUANTITY FROM PRODUCT
  WHERE P_QUANTITY = (SELECT MAX(P_QUANTITY) FROM PRODUCT);
```

- Here, the subquery returns only one value, so can be used anywhere a single value is expected (such as an operand to =)
- In general, a subquery can return
  - a single value
  - a column of data
  - a table of data

Types of Subqueries

- The subqueries we have seen so far execute once before the outer subquery is executed
- When this is the case, we have an uncorrelated subquery
- In contrast, a correlated subquery is a subquery that executes once for each row in the outer query
- This occurs when the inner query requires a value supplied by each row of the outer query

Types of Subqueries...: dml-correlated.sql

- Below, we show an uncorrelated subquery and an equivalent correlated subquery
- The correlated subquery uses `EXISTS`, which holds for those rows of the outer query for which the subquery contains at least one row

```
SELECT V_CODE, V_NAME FROM VENDOR
  WHERE V_CODE IN (SELECT V_CODE FROM PRODUCT);

SELECT V_CODE, V_NAME FROM VENDOR
  WHERE EXISTS (SELECT V_CODE FROM PRODUCT WHERE PRODUCT.V_CODE = VENDOR.V_CODE);
```

Views

- A view is a virtual table based on a select query
- The syntax for creating a view is below

```
CREATE VIEW VIEWNAME AS SELECT-QUERY
```

- You can use a view name anywhere a table name is expected
- Views are dynamically updated when their base tables change
- Views can provide security by restricting users to use specified columns or rows

Joins: dml-join.sql

```
SELECT * FROM VENDOR NATURAL JOIN PRODUCT;
SELECT V_NAME, P_DESCRIPT
  FROM VENDOR LEFT OUTER JOIN PRODUCT
  ON VENDOR.V_CODE = PRODUCT.V_CODE;
```

- To perform a JOIN, we include the type of join in the `FROM` clause
- If performing an inner or outer join, we also include the join conditions after `ON`
- Note that SQLite does not support RIGHT OUTER JOIN or FULL JOIN

Grouping Data: ddl-insert2.sql

- Let’s say we want to list each vendor in PRODUCT, and the P_QUANTITY of the vendor’s product with highest quantity
- We will repopulate the tables so we have more data to experiment with

```
DELETE FROM VENDOR;
DELETE FROM PRODUCT;

INSERT INTO VENDOR VALUES (1, ’Bryson, Inc.’, ’905’, ’1234567’);
INSERT INTO VENDOR (V_CODE, V_NAME, V_PHONE)
  VALUES (2, ’Smithson, Inc.’, ’7654321’);
INSERT INTO VENDOR (V_CODE, V_NAME, V_PHONE)
  VALUES (3, ’Danson, Inc.’, ’7257257’);

INSERT INTO PRODUCT VALUES (1, ’water bottle’, ’2008-10-20’, 40, 1);
INSERT INTO PRODUCT VALUES (2, ’baseball glove’, ’2007-03-14’, 9, 1);
INSERT INTO PRODUCT VALUES (3, ’laptop’, ’2008-10-22’, 3, 1);
INSERT INTO PRODUCT VALUES (4, ’spoon’, ’2004-05-04’, 6, 2);
INSERT INTO PRODUCT VALUES (5, ’baseball glove’, ’2007-03-14’, 12, 2);
INSERT INTO PRODUCT VALUES (6, ’television’, ’2008-10-22’, 3, NULL);
INSERT INTO PRODUCT VALUES (7, ’stove’, ’2008-03-27’, 1, NULL);
```

Grouping Data...: dml-groups.sql

- We will use MAX, but MAX operates on all of our rows, returning just one result
- Instead, we want MAX to operate on each vendor’s group of data separately
- We use GROUP BY for this

```
/*Doesn’t work*/
SELECT V_CODE, MAX (P_QUANTITY) FROM PRODUCT;

/*Works (GROUP BY creates groups for MAX)*/
SELECT V_CODE, MAX (P_QUANTITY) FROM PRODUCT GROUP BY V_CODE;
```

Operational Pipeline

From Owens, Definitive Guide to SQLite:

- It's helpful to understand the order in which things happen when an SQL SELECT executes
- Each clause except `FROM` takes the previous relation as input and produces a relation as output that it feeds to the next clause in the pipeline

1. Create initial relation in `FROM` clause, performing joins if present
2. `WHERE` executes, restricting that relation to rows we want
3. `GROUP BY` divides the rows into groups
4. `HAVING` filters groups (like `WHERE` filters `FROM`)
5. `ORDER BY` executes, reordering current results
6. `SELECT` projects out the columns we want
7. `DISTINCT` removes duplicate rows
8. `LIMIT` specifies the number of rows we want to return
9. `OFFSET` specifies where we want to begin returning rows

Set Operators

```
CREATE TABLE NEWPRODUCT (
  P_CODE TEXT NOT NULL UNIQUE,
  P_DESCRIPT TEXT NOT NULL CHECK (length(P_DESCRIPT) >= 5),
  P_INDATE TEXT NOT NULL,
  P_QUANTITY INTEGER NOT NULL,
  V_CODE INTEGER,
  PRIMARY KEY (P_CODE),
  FOREIGN KEY (V_CODE) REFERENCES VENDOR (V_CODE));

INSERT INTO NEWPRODUCT VALUES (3, 'laptop', '2008-10-22', 3, 1);
INSERT INTO NEWPRODUCT VALUES (8, 'textbook', '1994-05-25', 341, NULL);

SELECT * FROM PRODUCT UNION SELECT * FROM NEWPRODUCT;
```

- Recall the relational operators union, intersection, and difference
- These are implemented in SQL by combining two or more `SELECT`s with `UNION`, `INTERSECT` or `EXCEPT`
- Remember: the relations must be union-compatible
- We can also use `UNION ALL` to retain duplicate rows when we perform a union

SQL Functions: dml-functions.sql

- SQL functions allow you to process strings, perform mathematical operators, and work with dates and times
- Examples: `UPPER` converts a string to uppercase, `||` concatenates strings

```
SELECT P_DESCRIPT AS PRODUCT, UPPER(P_DESCRIPT) AS PRODUCT,
   P_DESCRIPT || " (" || V_NAME || ")" AS PRODUCT
  FROM PRODUCT NATURAL JOIN VENDOR;
```

SQL Functions...: dml-date.sql

- We have seen that DATE converts strings to dates
- We can use JULIANDAY to receive a date's Julian number (days that have passed between January 1, 4713 BC and the given date)
- Here is a query that returns the number of days each product has been in inventory

```
SELECT P_DESCRIPT, ROUND(JULIANDAY ('NOW') - JULIANDAY (P_INDATE)) AS DAYS_OLD
  FROM PRODUCT;
```