The SELECT Command, Aggregate Functions

Review and new SELECT options

The SELECT Command

- Used to extract data from a database
- One of the most frequently used commands in most applications
- We need to have a solid understanding of the SELECT command
- Is a ‘read only’ command – will not modify the database (unless embedded in a different command)

SQL Tables (2)

- Table body
  - Contains all of the rows
  - Each row consists of one data element for each column
  - All rows must have the same number of data elements (one for each column)
  - Each element can hold exactly one data value (or a NULL)
- SQL tables may hold duplicate rows
  - Duplicate rows are normally undesirable in practice

SQL Tables (3)

- Rows are unordered (the order of insertion has no meaning to the database)
  - Queries control the order in which they are displayed/manipulated

The SELECT Pipeline

- Tries to represent a generic framework to express many different types of queries
- SELECT has a large number of optional clauses (each with different options and formats)
- SQLite general format

```
SELECT [DISTINCT] columns
FROM table(s)
WHERE filter_expression
GROUP BY grouping_expressions
    HAVING filter_expression
GROUP BY ordering_expressions
LIMIT count
    OFFSET count
```

The SELECT Pipeline (2)

- The SELECT command must have a select heading (returns values)
- The rest of the lines are optional each representing an optional clause
- Each clause represent a step in the SELECT pipeline

SELECT DISTINCT

DISTINCT takes the result of the SELECT clause and filters out duplicate rows

```
sqlite> SELECT   REP_NUM   FROM   CUSTOMER;
            REP_NUM
            20
            35
            65
            35
            65
            20
            65
            35
            35
            20
```

```
sqlite> SELECT DISTINCT   REP_NUM   FROM   CUSTOMER;
            REP_NUM
            20
            35
            65
```

The SELECT clause pulls out just the REP_NUM column, and finally DISTINCT removes duplicate rows, reducing the number from 10 rows to 3 rows, all unique.

WHERE clause

```
sqlite> SELECT  customer_num, customer_name, rep_num  FROM  customer;

CUSTOMER_NUM    CUSTOMER_NAME                 REP_NUM
148             Al's Appliance and Sport      20
282             Brookings Direct              35
356             Ferguson's                    65
408             The Everything Shop           35
462             Bargains Galore               65
524             Kline's                       20
608             Johnson's Department Stor     65
687             Lee's Sport and Appliance     35
725             Deerfield's Four Seasons      35
842             All Season                    20
sqlite> SELECT  customer_num, customer_name, rep_num  FROM  customer
        WHERE rep_num = 20;

CUSTOMER_NUM    CUSTOMER_NAME                 REP_NUM
148             Al's Appliance and Sport      20
524             Kline's                       20
842             All Season                    20
```

WHERE clause (2)

```
sqlite> SELECT  customer_num, customer_name, rep_num  FROM  customer
            WHERE rep_num = 35;
```

Will produce:

```
CUSTOMER_NUM      CUSTOMER_NAME                        REP_NUM
282               Brookings Direct                     35
408               The Everything Shop                  35
687                Lee's Sport and Appliance           35
688                Deerfield's Four Seasons            35
```

```
sqlite> SELECT  customer_num, customer_name, rep_num  FROM  customer
            WHERE rep_num = 65;
```

Using WHERE to UPDATE and DELETE

```
sqlite> UPDATE customer SET state="GA" WHERE rep_num= 35;
```

(let’s make a copy of the customer table to practice deleting records)

```
sqlite> create table customer2 as select * from customer;
```

Delete all the records from the customer2 table for rep number 3**5**

```
sqlite> delete from customer2 where rep_num = 35;
```

Display all records from the customer2 table

```
sqlite> select * from customer2;
```

Aggregate Functions

- Take input and aggregate or combine all of column values from a group of rows to produce a single value

  count(), sum(), min(), max(), avg()

sqlite> select customer_num, rep_num, count(*) AS 'Number of Reps' from customer where rep_num=65;

<u>Will produce:</u>

```
CUSTOMER_NUM  REP_NUM     Number of Reps
608           65          3
```

Notice that if we choose to display the customer number, only the customer number for the last record matching this condition is displayed

<u>Examples:</u>

sqlite> select rep_num, count(*) AS 'Number of Reps' from customer where rep_num=65;

<u>Will produce:</u>

```
REP_NUM       Number of Reps
 65           3
```

Aggregate Functions (2)

Examples:

sqlite> select max(credit_limit) from customer;

Will produce:

max(credit_limit)

15000.0

sqlite> select min(credit_limit) from customer;

Will produce:

min(credit_limit)

5000.0

sqlite> select avg(credit_limit) AS 'Average Credit Limit' from customer;

Will produce:

Average Credit Limit

8500.0

GROUP BY

```
sqlite> SELECT customer_num, rep_num, count(*)
 FROM customer
 GROUP BY rep_num;
```

Will produce:

```
CUSTOMER_NUM   REP_NUM        count(*)
842            20             3
725            35             4
608            65             3
```

Again, notice that if we select the customer number, it only displays the last customer record for the last record where rep_num is 20, the customer record for the last record for the next rep_num, and so forth.

GROUP BY (2)

sqlite> select customer_num, balance, rep_num from customer;

Will produce:

```
CUSTOMER_NUM  BALANCE     REP_NUM
148           6550.0      20
282            431.5      35
356           5785.0      65
408           5285.25     35
462           3412.0      65
524          12762.0      20
608           2106.0      65
687           2851.0      35
725            248.0      35
842           8221.0      20
```

GROUP BY (3)

```
sqlite> rep_num, sum(balance) from customer where rep_num = 20;
```

Will produce:

```
 REP_NUM    sum(balance)
    20          27533.0
```

```
sqlite> rep_num, sum(balance) from customer group by rep_num;
```

Will produce:

```
REP_NUM    sum(balance)
20              27533.0
35                8815.75
65              11303.0
```

GROUP B (4)

- We could have three statements:

sqlite> select rep_num, count(*) from customer where rep_num = 20;

sqlite> select rep_num, count(*) from customer where rep_num = 35;

sqlite> select rep_num, count(*) from customer where rep_num = 65;

- Or, we could get the same results with a GROUP BY:

```
sqlite> select rep_num, count(*) from customer group by rep_num;
REP_NUM                count(*)
20                        3
35                        4
65                        3
```

HAVING

- Is placed after GROUP by to filter rows based on the results of the GROUP BY
- GROUP BY does all the work to create groups with like values
- HAVING filters the groups from GROUP BY in the same way that the WHERE clause filters rows from the FROM clause with one difference:
  - **WHERE** is expresses in terms of individual row values
  - **HAVING** is expressed in terms of aggregate values

```
sqlite> select rep_num , count(*) from customer group by rep_num;
REP_NUM          count(*)
20                  3
35                  4
65                  3
```

```
sqlite> select rep_num , count(*) from customer group by rep_num having count(*)  < 4;
REP_NUM              count(*)
20                  3
65                  3
```