SQL: miscellaneous

- subqueries, continued
- db administration
- constraints for tables
- window functions
- course evaluation

Recap: Subqueries

- performing operations in more than 1 step
- Subquery runs before the outer query, so that the outer query can use the result from the subquery as the underlying table.

Recap: Subqueries in FROM

- Example
  SELECT sub.*
  FROM (  
    SELECT *  
    FROM car  
    WHERE price < 35000  
  ) sub
  WHERE sub.battery = 24

Recap: Subqueries in WHERE

- Find the price of the second most expensive car
- we do this by first finding the highest price P, and then selecting the highest price that’s less than P
  SELECT MAX(price)
  FROM car
  WHERE price < (  
    SELECT MAX(price)  
    FROM car  
  )
Subqueries in JOIN

• Example: find the total sales (in all states) for each car model

```sql
SELECT car.model, sales
FROM car
JOIN (SELECT model, SUM(qty) AS sales
      FROM sales
      GROUP BY model
    ) sub
ON car.model = sub.model
```

Subqueries in JOIN

• The database runs the inner query/subquery:

```sql
SELECT model, SUM(qty) AS sales
FROM sales
GROUP BY model
```

• The inner query produces a result table

<table>
<thead>
<tr>
<th>model</th>
<th>sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>500E</td>
<td>30</td>
</tr>
<tr>
<td>I3</td>
<td>269</td>
</tr>
<tr>
<td>LEAF</td>
<td>897</td>
</tr>
<tr>
<td>560</td>
<td>18</td>
</tr>
<tr>
<td>SPARKEV</td>
<td>15</td>
</tr>
</tbody>
</table>

Subqueries in JOIN

• the subquery here has a name

```sql
...(
    SELECT model, SUM(qty) AS sales
    FROM sales
    GROUP BY model
) sub ...
```

Subqueries in JOIN

• After the inner query is done, the database runs the outer query using that result table produced from the inner query

```sql
SELECT car.model, sales
FROM car
JOIN sub
ON car.model = sub.model
```

<table>
<thead>
<tr>
<th>model</th>
<th>sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>500E</td>
<td>30</td>
</tr>
<tr>
<td>I3</td>
<td>269</td>
</tr>
<tr>
<td>LEAF</td>
<td>897</td>
</tr>
<tr>
<td>560</td>
<td>18</td>
</tr>
<tr>
<td>SPARKEV</td>
<td>15</td>
</tr>
</tbody>
</table>
Subqueries in JOIN

- joining that aggregated subquery is equal to joining then aggregating

```sql
SELECT car.model, SUM(sales.qty)
FROM car
JOIN sales
    ON car.model = sales.model
GROUP BY car.model
```

Other subqueries...

- Besides these common subqueries, there are other subqueries as well.
- All subqueries we see so far run only once. There are also subqueries that get to run more than once.
- Example coming up later today

Database Administration

- CRUD: create, read, update and delete
- can create, update and delete tables and records using SQL
- usually need special privileges to make changes to the database

- Privileges can be:
  - Select
  - Insert
  - Delete
  - References (cols) - allow to create a foreign key that references the specified column(s)
  - All
- Privileges can later be revoked
- Users can be single users or group
### Deleting a table

**DROP TABLE** table_name

- example: if these tables exist, drop them

```
DROP TABLE IF EXISTS CAR, SALES, STATE;
```

### Creating a table

**CREATE TABLE** table_name(

- column_name data_type(size) constraints,
- column_name2 data_type(size) constraints, ...

```
);  
```

- each table must have a name so that it can be referenced

- example

```
CREATE TABLE CAR (  
    MODEL VARCHAR(10) PRIMARY KEY,  
    MAN VARCHAR(10) NOT NULL,  
    PRICE NUMERIC, ...  
);  
```

### Constraints Overview

- **NOT NULL** - a column cannot store NULL value
- **UNIQUE** - each row for a column must have a unique value
- **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Ensures that a column (or combination of two or more columns) have a unique identity
- **FOREIGN KEY** - Ensure the referential integrity of the data in one table to match values in another table
- **CHECK** - Ensures that the value in a column meets a specific condition
- **DEFAULT** - a default value for a column

### Changing a table

- Adding/deleting a new column
- changing the data type of a column

- use **ALTER TABLE**, but syntaxes differ by implementation
Deleting a record

```
DELETE FROM table_name
WHERE condition;
• The WHERE clause specifies which records
  should be deleted
• if WHERE is not present, all records will be
  deleted
```

Adding a record

```
INSERT INTO table_name
VALUES (value1,value2,value3,...);
OR
INSERT INTO table_name
(column1,column2,column3,...)
VALUES (value1,value2,value3,...);
• Can either specify the column name or not
• example
INSERT INTO CAR
(MODEL, MAN, PRICE, BATTERY, MOTORKW, MPGE, MILES, WEIGHT)
VALUES ('I3', 'BMW',
42375, 22, 125, 123, 80, 2700);
```

Updating a record

```
UPDATE table_name
SET column1=value1,column2=value2,...
WHERE condition;
• Again, the WHERE clause specifies which
  records should be deleted
• if WHERE is not present, all records will be
  updated
```

Updating a record

```
• Can combine delete and update statement with
  other clauses
• example: update the price of cars that are not sold
  in any state
UPDATE car
SET price = 30000
WHERE model NOT IN
(SELECT DISTINCT model
 FROM state);
<-- all models sold
in all states
```
Constraints

- also called **integrity constraints**
- come from the semantics from real world
- inserts/deletes/updates that violate the constraints will be disallowed
- prevent inconsistencies in the database (e.g. car model has to be a string, price has to > 0)
- Types of constraints: domain constraints, primary key constraints, foreign key constraints, general constraints

Domain Constraints, General Constraints

- domain constraints: field values must be of right type. Always enforced.
- general constraints: checked every time when a record is inserted or updated

```
CREATE TABLE CAR (  
    MODEL     VARCHAR(10) PRIMARY KEY,
    MAN       VARCHAR(10)  NOT NULL,
    PRICE     NUMERIC,
    CONSTRAINT price_chk CHECK (PRICE > 0 AND PRICE < 100000)
);
```

General Constraints

- general constraints can be named

```
CREATE TABLE CAR (  
    MODEL     VARCHAR(10) PRIMARY KEY,
    MAN       VARCHAR(10)  NOT NULL,
    PRICE     NUMERIC,
    CONSTRAINT price_chk CHECK (PRICE > 0 AND PRICE < 100000)
);
```

- can drop a check constraint as well

Key Constraints

- the most common type of constraints
- a way to associate tuples in different relations
- something the database administrator specifies, and they’re there to enforce constraints

```
<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>Terry</td>
<td>3.4</td>
</tr>
<tr>
<td>10002</td>
<td>Elena</td>
<td>3.9</td>
</tr>
<tr>
<td>10003</td>
<td>Kevin</td>
<td>2.8</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>English101</td>
<td>C</td>
</tr>
<tr>
<td>10002</td>
<td>History53</td>
<td>B</td>
</tr>
<tr>
<td>10003</td>
<td>Math102</td>
<td>B</td>
</tr>
<tr>
<td>10001</td>
<td>Music203</td>
<td>A</td>
</tr>
</tbody>
</table>
Primary Key Constraints

- the primary key is used to uniquely identify each record in the table
  - all values are UNIQUE
  - cannot be NULL
  - each table only needs 1 primary key (some needs no primary keys, but they usually need one)
  - can be combinations of multiple fields
- Examples: selecting a primary key
  - sid could be the primary key for Students
  - Enrollment doesn’t have a primary key yet
  - name usually can’t be selected to be the primary key of Students

Primary Key Constraints

- How should we enforce this constraint? “There should be only 1 single grade given a student and a course.”

CREATE TABLE Enrollment
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid, cid));

CREATE TABLE Enrollment
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY sid,
UNIQUE (cid, grade));

Primary Key Constraints

What if I run the following two queries for each of the tables?

INSERT INTO Enrollment
VALUES (10001, Math101, F);

INSERT INTO Enrollment
VALUES (10001, Math101, A+);

Primary Key Constraints

What if I run the following two queries for each of the tables?

INSERT INTO Enrollment
VALUES (10001, Math101, F);

INSERT INTO Enrollment
VALUES (10001, History115, A+);
Primary Key Constraints

- How should I enforce this constraint? “There should be only 1 single grade given a student and a course.”

```
CREATE TABLE Enrollment
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid,cid));
```

What if I run the following two queries for each of the tables?

```
INSERT INTO Enrollment
VALUES (10001,
       Math101, A);
```

```
CREATE TABLE Enrollment
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid,cid),
UNIQUE (cid, grade));
```

```
INSERT INTO Enrollment
VALUES (10002,
       Math101, A);
```

Foreign Key Constraints

- Foreign key: a set of fields in a tuple in one relation that refer to a tuple in another relation
- The foreign key constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

```
CREATE TABLE Enrollment
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid,cid),
FOREIGN KEY (sid) REFERENCES Students);
```

Students

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>Terry</td>
<td>3.4</td>
</tr>
<tr>
<td>10002</td>
<td>Elena</td>
<td>3.9</td>
</tr>
<tr>
<td>10003</td>
<td>Kevin</td>
<td>2.8</td>
</tr>
</tbody>
</table>

Primary Key

Enrollment

<table>
<thead>
<tr>
<th>sid</th>
<th>cname</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>English 101</td>
<td>C</td>
</tr>
<tr>
<td>10002</td>
<td>History 53</td>
<td>B</td>
</tr>
<tr>
<td>10003</td>
<td>Math 102</td>
<td>B</td>
</tr>
<tr>
<td>10001</td>
<td>Music 203</td>
<td>A</td>
</tr>
</tbody>
</table>
Foreign Key Constraints

• What if we want to insert sid=99999, cname=Astro105, grade=B into the Enrollment table?
• What if we delete student with sid=10001 from a Students table?
  • delete corresponding records from Enrollment?
  • set sid in Enrollment to the DEFAULT value?
  • set sid in Enrollment to NULL?
• What if we update the student with sid=10001 to have the sid to be 11111? (updating a primary key field)

<table>
<thead>
<tr>
<th>Students</th>
<th>Enrollment</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>10001</td>
<td>Terry</td>
</tr>
<tr>
<td>10002</td>
<td>Elena</td>
</tr>
<tr>
<td>10003</td>
<td>Kevin</td>
</tr>
<tr>
<td>10001</td>
<td>Music</td>
</tr>
</tbody>
</table>

Window Functions

• So far, all calculations are ‘stateless’ — each row is calculated independently of each other, and the results are not affected by the ordering of the records.
• The window functions are different because they are able to access more than just the current row of the query result.
• The type of calculation that window functions do is like an aggregate function, but does not cause rows to become grouped into a single output row — the rows retain their separate identities.

http://www.postgresql.org/docs/9.1/static/tutorial-window.html

Window Functions

• Warning: window functions are not implemented in mySQL. (can’t try them out here)
• But lots of other implementations support window functions, e.g. PostgreSQL

<window function> OVER
  ([PARTITION BY <expression list>]
  [ORDER BY <expression [ASC|DESC] list>])

Window Functions

• Example: the running total

```sql
SELECT model, qty, 
  SUM(qty) OVER (ORDER BY model) 
  AS running_total 
FROM sales
```
Window Functions

```sql
SELECT model, qty, SUM(qty) OVER (ORDER BY model) AS running_total
FROM sales
```

- SUM(qty) is an aggregate function
- OVER indicates a window function
- ORDER BY says that the running total should be calculated in that order (in this case alphabetical order by the model name)

---

Window Functions

```sql
SELECT model, qty, SUM(qty) OVER (PARTITION BY model ORDER BY qty) AS running_total
FROM sales
```

- PARTITION BY says that the running total should be calculated by each partition
- when we get to each model, the running total gets reset
- the order within each partition is determined by the qty field

---

Window Functions

<window function> OVER

```sql
([[PARTITION BY <expression list>]
[ORDER BY <expression [ASC|DESC] list>])
```

- Functions that can be used as window functions:
  - the normal aggregate functions: SUM, AVG, COUNT
  - ROW_NUMBER: the row number of a row
  - RANK, DENSE_RANK: gives the rank of a row compared with other rows
  - LAG, LEAD: get values after and before the current row
  - ...and more!

---

Window Functions

- Example: ranking
  - I want to find the rankings of the cars by their prices. If there’s a tie for price, they should have the same ranking.

<table>
<thead>
<tr>
<th>model</th>
<th>price</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>S60</td>
<td>71070</td>
<td>1</td>
</tr>
<tr>
<td>BCE</td>
<td>42375</td>
<td>2</td>
</tr>
<tr>
<td>LEAF</td>
<td>27495</td>
<td>5</td>
</tr>
<tr>
<td>LEAF</td>
<td>31820</td>
<td>4</td>
</tr>
<tr>
<td>LEAF</td>
<td>32650</td>
<td>3</td>
</tr>
<tr>
<td>S60</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>SPARKEV</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

---
Window Functions

• find the ranking of the car by their prices

```sql
SELECT model, price,
       DENSE_RANK() OVER (ORDER BY price)
AS rank
FROM car
ORDER BY price DESC
```

• In the tie, DENSE_RANK does not skip ranks, and RANK() skips ranks

• There are ways to get rankings without window functions — using subqueries or joins!

Example: ranking

• idea1: for each price p, count the number of distinct prices greater or equal to p.

```sql
SELECT model, price,
       (SELECT COUNT(DISTINCT price)
        FROM car c2
        WHERE c2.price >= c1.price
       ) AS rank
FROM car c1
ORDER BY price DESC
```

Example: ranking

• There are two tables c1 and c2.

• The database goes through each price in c1, and run the inner query to count the number of distinct prices >= this current price — which is also the rank.

• The inner query gets to run 6 times!

<table>
<thead>
<tr>
<th>MODEL</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I3</td>
<td>42375</td>
</tr>
<tr>
<td>BCE</td>
<td>42375</td>
</tr>
<tr>
<td>SPARKEV</td>
<td>27495</td>
</tr>
<tr>
<td>LEAF</td>
<td>31820</td>
</tr>
<tr>
<td>S60</td>
<td>71070</td>
</tr>
<tr>
<td>500E</td>
<td>32650</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MODEL</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I3</td>
<td>42375</td>
</tr>
<tr>
<td>BCE</td>
<td>42375</td>
</tr>
<tr>
<td>SPARKEV</td>
<td>27495</td>
</tr>
<tr>
<td>LEAF</td>
<td>31820</td>
</tr>
<tr>
<td>S60</td>
<td>71070</td>
</tr>
<tr>
<td>500E</td>
<td>32650</td>
</tr>
</tbody>
</table>

Example: ranking

• idea2: for each price p, match with all prices greater than p, and count the number of those greater prices

```sql
SELECT c1.model, c1.price,
       COUNT(DISTINCT c2.price) AS rank
FROM car c1
JOIN car c2
ON c1.price <= c2.price
GROUP BY c1.model, c1.price
ORDER BY c1.price DESC
```
In conclusion...

• Subquery runs before the outer query, so that the outer query can use the result from the subquery as the underlying table.
• Able to create, update and delete tables and records in SQL.
• Integrity constraints prevents inconsistencies in the database.
• A window function performs a calculation across a set of table rows that are somehow related to the current row.

The big ideas

• The databases we deal with follows the relational model
  • Data is represented as a collection of two dimensional tables
  • Values of attributes are drawn from a domain
• SQL provides functionality to do CRUD (create, read, update and delete)
• Usually there are many ways to write a query that does the same thing