SQL

April 25th, 2002

Agenda

• Grouping and aggregation
• Sub-queries
• Updating the database
• Views
• More on views

Union, Intersection, Difference

(SELECT name
FROM Person
WHERE City="Seattle")

UNION

(SELECT name
FROM Person, Purchase
WHERE buyer=name AND store="The Bon")

Similarly, you can use INTERSECT and EXCEPT. You must have the same attribute names (otherwise: rename).

Aggregation

SQL supports several aggregation operations:

SUM, MIN, MAX, AVG, COUNT

Aggregation: Count

SELECT Sum(price)
FROM Product
WHERE maker="Toyota"

COUNT applies to duplicates, unless otherwise stated:

SELECT Count(name, category) same as Count(*)
FROM Product
WHERE year > 1995

Better:

SELECT Count(DISTINCT name, category)
FROM Product
WHERE year > 1995

Except COUNT, all aggregations apply to a single attribute
Simple Aggregation

Purchase(product, date, price, quantity)

Example 1: find total sales for the entire database
SELECT Sum(price * quantity)
FROM Purchase

Example 1': find total sales of bagels
SELECT Sum(price * quantity)
FROM Purchase
WHERE product = 'bagel'

Simple Aggregations

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>0.85</td>
<td>15</td>
</tr>
<tr>
<td>Banana</td>
<td>10/22</td>
<td>0.52</td>
<td>7</td>
</tr>
<tr>
<td>Banana</td>
<td>10/19</td>
<td>0.52</td>
<td>17</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/20</td>
<td>0.85</td>
<td>20</td>
</tr>
</tbody>
</table>

Grouping and Aggregation

Usually, we want aggregations on certain parts of the relation.

Buyer(product, date, price, quantity)

Example 2: find total sales after 9/1 per product.
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > "9/1"
GROUPBY product

Grouping and Aggregation

1. Compute the relation (I.e., the FROM and WHERE).
2. Group by the attributes in the GROUPBY
3. Select one tuple for every group (and apply aggregation)

SELECT can have (1) grouped attributes or (2) aggregates.

First compute the relation (date > "9/1") then group by product:

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>10/19</td>
<td>0.52</td>
<td>17</td>
</tr>
<tr>
<td>Banana</td>
<td>10/22</td>
<td>0.52</td>
<td>7</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/20</td>
<td>0.85</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>0.85</td>
<td>15</td>
</tr>
</tbody>
</table>

Then, aggregate

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>$29.75</td>
</tr>
<tr>
<td>Banana</td>
<td>$12.48</td>
</tr>
</tbody>
</table>

SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > "9/1"
GROUPBY product
Another Example

<table>
<thead>
<tr>
<th>Product</th>
<th>SumSales</th>
<th>MaxQuantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>$12.48</td>
<td>17</td>
</tr>
<tr>
<td>Bagel</td>
<td>$29.75</td>
<td>20</td>
</tr>
</tbody>
</table>

For every product, what is the total sales and max quantity sold?

\[
\text{SELECT product, Sum(price * quantity) AS SumSales, Max(quantity) AS MaxQuantity}
\]

\[
\text{FROM Purchase}
\]

\[
\text{GROUP BY product}
\]

HAVING Clause

Same query, except that we consider only products that had at least 100 buyers.

\[
\text{SELECT product, Sum(price * quantity)}
\]

\[
\text{FROM Purchase}
\]

\[
\text{WHERE date > "9/1"}
\]

\[
\text{GROUP BY product}
\]

\[
\text{HAVING Sum(quantity) > 30}
\]

HAVING clause contains conditions on aggregates.

General form of Grouping and Aggregation

\[
\text{SELECT S}
\]

\[
\text{FROM R}_1, \ldots, R_n
\]

\[
\text{WHERE C}_1
\]

\[
\text{GROUP BY a}_1, \ldots, a_k
\]

\[
\text{HAVING C}_2
\]

- S = may contain attributes a\_1, a\_k and/or any aggregates but NO OTHER ATTRIBUTES
- C\_1 = is any condition on the attributes in R\_1, \ldots, R\_n
- C\_2 = is any condition on aggregate expressions

Evaluation steps:
1. Compute the FROM-WHERE part, obtain a table with all attributes in R\_1, \ldots, R\_n
2. Group by the attributes a\_1, \ldots, a\_k
3. Compute the aggregates in C\_2 and keep only groups satisfying C\_2
4. Compute aggregates in S and return the result

Aggregation

Author(login, name)
Document(url, title)
Wrote(login, url)
Mentions(url, word)

- Find all authors who wrote at least 10 documents:

\[
\text{SELECT author.name}
\]

\[
\text{From author, wrote}
\]

\[
\text{Where author.login=wrote.login}
\]

\[
\text{Groupby author.name}
\]

\[
\text{Having count(wrote.url) > 10}
\]
• Find all authors who have a vocabulary over 10000:

```sql
SELECT author.name
FROM author, wrote, mentions
WHERE author.login = wrote.login AND wrote.url = mentions.url
GROUP BY author.name
HAVING count(distinct mentions.word) > 10000
```

### Exercises

- **Ex #1:** Find people who bought telephony products.
- **Ex #2:** Find names of people who bought American products.
- **Ex #3:** Find names of people who bought American products and did not buy French products.
- **Ex #4:** How much money did Fred spend on purchases?
- **Ex #5:** What is the number and sum of the product sales by country of origin?

### Subqueries

A subquery producing a single tuple:

```sql
SELECT Purchase.product
FROM Purchase
WHERE buyer = (SELECT name
FROM Person
WHERE ssn = "123456789");
```

In this case, the subquery returns one value.

If it returns more, it's a run-time error.

Can we express this query without a subquery?

### Subqueries Returning Relations

Find companies who manufacture products bought by Joe Blow.

```sql
SELECT Company.name
FROM Company, Product
WHERE Company.name = maker
AND Product.name = (SELECT product
FROM Purchase
WHERE buyer = "Joe Blow");
```

Here the subquery returns a set of values.

### Subqueries Returning Relations

Equivalent to:

```sql
SELECT Company.name
FROM Company, Product, Purchase
WHERE Company.name = maker
AND Product.name = product
AND buyer = "Joe Blow"
```

Is this query equivalent to the previous one?

---

You can also use:

```sql
s > ALL R
s > ANY R
EXISTS R
```

Find products that are more expensive than all those produced by "Gizmo-Works."

```sql
SELECT name
FROM Product
WHERE price > ALL (SELECT price
FROM Purchase
WHERE maker = "Gizmo-Works");
```
Question for Database Fans and their Friends

- Can we express this query as a single SELECT-FROM-WHERE query, without subqueries?

- Hint: show that all SFW queries are monotone (figure out what this means). A query with ALL is not monotone

Conditions on Tuples

SELECT Company.name
FROM Company, Product
WHERE Company.name=maker
AND (Product.name, price) IN
(SELECT product, price)
FROM Purchase
WHERE buyer = "Joe Smith";

Correlated Queries

Movie (title, year, director, length)
Find movies whose title appears more than once.

SELECT title
FROM Movie AS x
WHERE year < ANY
(SELECT year
FROM Movie
WHERE title = x.title);

Note (1) scope of variables (2) this can still be expressed as single SFW correlation

Complex Correlated Query

Product (pname, price, category, maker, year)
- Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

SELECT pname, maker
FROM Product AS x
WHERE price > ALL (SELECT price
FROM Product AS y
WHERE x.maker = y.maker AND y.year < 1972);

Powerful, but much harder to optimize!

Removing Duplicates

SELECT DISTINCT Company.name
FROM Company, Product
WHERE Company.name=maker
AND (Product.name, price) IN
(SELECT product, price)
FROM Purchase
WHERE buyer = "Joe Blow"

Conserving Duplicates

The UNION, INTERSECTION and EXCEPT operators operate as sets, not bags.

(SELECT name
FROM Person
WHERE City="Seattle")
UNION ALL

(SELECT name
FROM Person, Purchase
WHERE buyer=name AND store="The Bon")
Modifying the Database

Three kinds of modifications
- Insertions
- Deletions
- Updates

Sometimes they are all called “updates”

Insertions

General form:

\[
\text{INSERT INTO } R(A_1, \ldots, A_n) \text{ VALUES } (v_1, \ldots, v_n)
\]

Example: Insert a new purchase to the database:

\[
\text{INSERT INTO Purchase(buyer, seller, product, store)}
\]

VALUES ('Joe', 'Fred', 'wakeup-clock-espresso-machine', 'The Sharper Image')

May drop attribute names if give them in order.

Insertion: an Example

The query replaces the VALUES keyword.
Here we insert *many* tuples into PRODUCT

Product\(\{\text{name, listPrice, category}\}\)
Purchase\(\{\text{prodName, buyerName, price}\}\)

prodName is foreign key in Product.name

Suppose database got corrupted and we need to fix it:

Task: insert in Product all prodNames from Purchase

<table>
<thead>
<tr>
<th>prodName</th>
<th>category</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>gadget</td>
<td>gadgets</td>
<td>225</td>
</tr>
<tr>
<td>camera</td>
<td>Smith</td>
<td>200</td>
</tr>
<tr>
<td>camera</td>
<td>Smith</td>
<td>225</td>
</tr>
</tbody>
</table>

Insertion: an Example

depends on the implementation
Deletions

Example:

DELETE FROM PURCHASE
WHERE seller = 'Joe' AND product = 'Brooklyn Bridge';

Factoid about SQL: there is no way to delete only a single occurrence of a tuple that appears twice in a relation.

Updates

Example:

UPDATE PRODUCT
SET price = price/2
WHERE Product.name IN (SELECT product FROM Purchase WHERE Date = 'Oct, 25, 1999');

Data Definition in SQL

So far we have seen the Data Manipulation Language, DML. Next: Data Definition Language (DDL)

Data types:
- Defines the types.

Data definition: defining the schema.
- Create tables
- Delete tables
- Modify table schema

Indexes: to improve performance

Data Types in SQL

- Character strings (fixed or varying length)
- Bit strings (fixed or varying length)
- Integer (SHORTINT)
- Floating Point
- Dates and times

Domains (=types) will be used in table declarations.

To reuse domains:
CREATE DOMAIN address AS VARCHAR(55)

Creating Tables

Example:

CREATE TABLE Person(
    name VARCHAR(30),
    social-security-number INTEGER,
    age SHORTINT,
    city VARCHAR(30),
    gender BIT(1),
    Birthdate DATE
);

Deleting or Modifying a Table

Deleting: Example: DROP Person;

Altering: (adding or removing an attribute).

Example:

ALTER TABLE Person
ADD phone CHAR(16);
ALTER TABLE Person
DROP age;

What happens when you make changes to the schema?
Default Values

Specifying default values:

```sql
CREATE TABLE Person(
    name VARCHAR(30),
    social-security-number INTEGER,
    age SHORTINT DEFAULT 100,
    city VARCHAR(30) DEFAULT 'Seattle',
    gender CHAR(1) DEFAULT '?',
    Birthdate DATE
);```

The default of defaults: NULL

Indexes

REALLY important to speed up query processing time.

Suppose we have a relation

Person (name, age, city)

Sequential scan of the file Person may take long:

```
SELECT * FROM Person WHERE name = "Smith"
```

Creating Indexes

Syntax:

```
CREATE INDEX nameIndex ON Person(name)
```

Indexes can be useful in range queries too:

```
CREATE INDEX ageIndex ON Person(age)
```

Creating Indexes

Indexes can be created on more than one attribute:

Example:

```
CREATE INDEX doubleIndex ON Person (age, city)
```

Helps in:

```
SELECT * FROM Person WHERE age = 55 AND city = "Seattle"
```

But not in:

```
SELECT * FROM Person WHERE city = "Seattle"
```

Indexes

Indexes can be created on more than one attribute:

Example:

```
CREATE INDEX doubleIndex ON Person (age, city)
```

Helps in:

```
SELECT * FROM Person WHERE age = 55 AND city = "Seattle"
```

But not in:

```
SELECT * FROM Person WHERE city = "Seattle"
```
Defining Views

Views are relations, except that they are not physically stored.

For presenting different information to different users

Employee(ssn, name, department, project, salary)

CREATE VIEW Developers AS
SELECT name, project
FROM Employee
WHERE department = "Development"

Payroll has access to Employee, others only to Developers

A Different View

Person(name, city)
Purchase(buyer, seller, product, store)
Product(name, maker, category)

CREATE VIEW Seattle-view AS
SELECT buyer, seller, product, store
FROM Person, Purchase
WHERE Person.city = "Seattle" AND
Person.name = Purchase.buyer

We have a new virtual table:
Seattle-view(buyer, seller, product, store)

A Different View

We can later use the view:

SELECT name, store
FROM Seattle-view, Product
WHERE Seattle-view.product = Product.name AND
Product.category = "shoes"

What Happens When We Query a View?

SELECT name, Seattle-view.store
FROM Seattle-view, Product
WHERE Seattle-view.product = Product.name AND
Product.category = "shoes"

SELECT name, Purchase.store
FROM Person, Purchase, Product
WHERE Person.city = "Seattle" AND
Person.name = Purchase.buyer AND
Purchase.product = Product.name AND
Product.category = "shoes"

Types of Views

- Virtual views:
  - Used in databases
  - Computed only on-demand – slow at runtime
  - Always up to date
- Materialized views
  - Used in data warehouses (but recently also in DBMS)
  - Precomputed offline – fast at runtime
  - May have stale data

Updating Views

How can I insert a tuple into a table that doesn’t exist?

Employee(son, name, department, project, salary)

CREATE VIEW Developers AS
SELECT name, project
FROM Employee
WHERE department = “Development”

If we make the following insertion:

INSERT INTO Developers
VALUES(“Joe”, “Optimizer”)

It becomes:

INSERT INTO Employee
**Non-Updatable Views**

```
CREATE VIEW Seattle-view AS
SELECT seller, product, store
FROM Person, Purchase
WHERE Person.city = "Seattle" AND
      Person.name = Purchase.buyer
```

How can we add the following tuple to the view?

("Joe", "Shoe Model 12345", "Nine West")

What do we put in the Person.name and Purchase.buyer columns?

---

**Answering Queries Using Views**

- What if we want to use a set of views to answer a query.
- Why?
  - The obvious reason…
  - Answering queries over web data sources.
- Very cool stuff! (i.e., I did a lot of research on this).

---

**Reusing a Materialized View**

- Suppose I have only the result of SeattleView:
  ```
  SELECT buyer, seller, product, store
  FROM Person, Purchase
  WHERE Person.city = 'Seattle' AND
        Person.name = Purchase.buyer
  ```
  and I want to answer the query
  ```
  SELECT buyer, seller
  FROM Person, Purchase
  WHERE Person.city = 'Seattle' AND
        Person.name = Purchase.buyer AND
        Purchase.product = 'gizmo'.
  ```
  Then, I can rewrite the query using the view.

---

**Query Rewriting Using Views**

Rewritten query:

```
SELECT buyer, seller
FROM SeattleView
WHERE product = 'gizmo'
```

Original query:

```
SELECT buyer, seller
FROM Person, Purchase
WHERE Person.city = 'Seattle' AND
      Person.name = Purchase.buyer AND
      Purchase.product = 'gizmo'.
```

---

**Another Example**

- I still have only the result of SeattleView:
  ```
  SELECT buyer, seller, product, store
  FROM Person, Purchase
  WHERE Person.city = 'Seattle' AND
        Person.name = Purchase.buyer
  ```
  but I want to answer the query
  ```
  SELECT buyer, seller
  FROM Person, Purchase
  WHERE Person.city = 'Seattle' AND
        Person.name = Purchase.buyer AND
        Person.Phone LIKE '206 543 %'.
  ```

---

**And Now?**

- I still have only the result of SeattleView:
  ```
  SELECT buyer, seller, product, store
  FROM Person, Purchase
  WHERE Person.city = 'Seattle' AND
        Person.name = Purchase.buyer
  ```
  but I want to answer the query
  ```
  SELECT buyer, seller
  FROM Person, Purchase, Product
  WHERE Person.city = 'Seattle' AND
        Person.name = Purchase.buyer AND
        Purchase.product = Product.name.
  ```
And Now?

- I still have only the result of:
  ```sql
  SELECT seller, buyer, Sum(Price)
  FROM Purchase
  WHERE Purchase.store = 'The Bon'
  GROUP BY seller, buyer
  ```

- but I want to answer the query
  ```sql
  SELECT seller, Sum(Price)
  FROM Purchase
  WHERE Person.store = 'The Bon'
  GROUP BY seller
  ```

And what if it’s the other way around?

Finally...

- I still have only the result of:
  ```sql
  SELECT seller, buyer, Count(*)
  FROM Purchase
  WHERE Purchase.store = 'The Bon'
  GROUP BY seller, buyer
  ```

- but I want to answer the query
  ```sql
  SELECT seller, Count(*)
  FROM Purchase
  WHERE Person.store = 'The Bon'
  GROUP BY seller
  ```

The General Problem

- Given a set of views V1, ..., Vn, and a query Q, can we answer Q using only the answers to V1, ..., Vn?

- Why do we care?
  - We can answer queries more efficiently.
  - We can query data sources on the WWW in a principled manner.

- Many, many papers on this problem.

- The best performing algorithm: The MiniCon Algorithm, (Pottinger & (Ha)Levy, 2000).

Querying the WWW

- Assume a virtual schema of the WWW, e.g.,
  ```
  Course(number, university, title, prof, quarter)
  ```

- Every data source on the web contains the answer to a view over the virtual schema:
  ```sql
  UW database: SELECT number, title, prof
  FROM Course
  WHERE univ='UW' AND quarter='2/02'
  ```

  ```sql
  Stanford database: SELECT number, title, prof, quarter
  FROM Course
  WHERE univ='Stanford'
  ```

- User query: find all professors who teach “database systems”

Null Values and Outerjoins

- If x=Null then 4*(3-x)/7 is still NULL

- If x=Null then x="Joe" is UNKNOWN

- Three boolean values:
  - FALSE = 0
  - UNKNOWN = 0.5
  - TRUE = 1

Null Values and Outerjoins

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

- Rule in SQL: include only tuples that yield TRUE

SELECT *
FROM Person
WHERE (age < 25) AND
    (height > 6 OR weight > 190)
Null Values and Outerjoins

Unexpected behavior:

```sql
SELECT *
FROM Person
WHERE age < 25 OR age >= 25
```

Some Persons are not included!

Null Values and Outerjoins

Can test for NULL explicitly:

- `x IS NULL`
- `x IS NOT NULL`

```sql
SELECT *
FROM Person
WHERE age < 25 OR age >= 25 OR age IS NULL
```

Now it includes all Persons

Null Values and Outerjoins

Explicit joins in SQL:

```
Product(name, category)
Purchase(prodName, store)
```

```sql
SELECT Product.name, Purchase.store
FROM Product JOIN Purchase ON Product.name = Purchase.prodName
```

Same as:

```sql
SELECT Product.name, Purchase.store
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
```

But Products that never sold will be lost!

Null Values and Outerjoins

Left outer joins in SQL:

```
Product(name, category)
Purchase(prodName, store)
```

```sql
SELECT Product.name, Purchase.store
FROM Product LEFT OUTER JOIN Purchase ON Product.name = Purchase.prodName
```

Outer Joins

- Left outer join:
  - Include the left tuple even if there’s no match

- Right outer join:
  - Include the right tuple even if there’s no match

- Full outer join:
  - Include the both left and right tuples even if there’s no match
SQL: Constraints and Triggers

- Chapter 6 Ullman and Widom
- Certain properties we’d like our database to hold
- Modification of the database may break these properties
- Build handlers into the database definition
- Key constraints
- Referential integrity constraints.

Declaring a Primary Keys in SQL

```
CREATE TABLE MovieStar (  
    name CHAR(30) PRIMARY KEY,  
    address VARCHAR(255),  
    gender CHAR(1));
```

OR:

```
CREATE TABLE MovieStar (  
    name CHAR(30),  
    address VARCHAR(255),  
    gender CHAR(1)  
    PRIMARY KEY (name));
```

Primary Keys with Multiple Attributes

```
CREATE TABLE MovieStar (  
    name CHAR(30),  
    address VARCHAR(255),  
    gender CHAR(1),  
    PRIMARY KEY (name, address));
```

Other Keys

```
CREATE TABLE MovieStar (  
    name CHAR(30),  
    address VARCHAR(255),  
    phone CHAR(10) UNIQUE,  
    gender CHAR(1),  
    petName CHAR(50),  
    PRIMARY KEY (name),  
    UNIQUE (gender, petName));
```

Foreign Key Constraints

```
CREATE TABLE ActedIn (  
    Name CHAR(30) PRIMARY KEY,  
    MovieName CHAR(30)  
    REFERENCES Movies(MovieName),  
    Year INT);  
```

OR

```
CREATE TABLE ActedIn (  
    Name CHAR(30) PRIMARY KEY,  
    MovieName CHAR(30),  
    Year INT,  
    FOREIGN KEY MovieName  
    REFERENCES Movies(MovieName)  
    REFERENCES Movies(MovieName)  
    PRIMARY KEY
```

• MovieName must be a PRIMARY KEY
How do we Maintain them?

- Given a change to DB, there are several possible violations:
  - Insert new tuple with bogus foreign key value
  - Update a tuple to a bogus foreign key value
  - Delete a tuple in the referenced table with the referenced foreign key value
  - Update a tuple in the referenced table that changes the referenced foreign key value

How to Maintain?

- Recall, ActedIn has FK MovieName...
  
  Movies(MovieName, year)
  (Fatal Attraction, 1987)

  ActedIn(ActorName, MovieName)
  (Michael Douglas, Fatal Attraction)

  insert: (Rick Moranis, Strange Brew)

  Policies for handling the change…
  - Reject the update (default)
  - Cascade (example: cascading deletes)
  - Set NULL

  Can set update and delete actions independently in CREATE TABLE

  MovieName CHAR(30)
  REFERENCES Movies(MovieName))
  ON DELETE SET NULL
  ON UPDATE CASCADE