Advanced SQL

Lecture 3

Outline

• Unions, intersections, differences
• Subqueries, Aggregations, NULLs
• Modifying databases, Indexes, Views

Reading:
• Textbook chapters 6.2 and 6.3
• from “SQL for Nerds”: chapter 4, “More complex queries” (you will find it very useful for subqueries)
• Pointbase developer manual
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).

Conserving Duplicates

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

UNION ALL

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

A subquery producing a single value:

```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.

Subqueries

Can say the same thing without a subquery:

```sql
SELECT Purchase.product
FROM   Purchase, Person
WHERE  buyer = name AND ssn = '123456789'
```

This is equivalent to the previous one when the ssn is a key and ‘123456789’ exists in the database; otherwise they are different.
Subqueries Returning Relations

Find companies that manufacture products bought by Joe Blow.

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

Here the subquery returns a set of values: no more runtime errors.

Equivalent to:

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

Is this query equivalent to the previous one ?

Beware of duplicates !
Removing Duplicates

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

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

Now they are equivalent

Subqueries Returning Relations

You can also use:

- \( s > \text{ALL} \ R \)
- \( s > \text{ANY} \ R \)
- \( \text{EXISTS} \ R \)

Product (\( \text{pname, price, category, maker} \))

Find products that are more expensive than all those produced by “Gizmo-Works”

```sql
SELECT name
FROM Product
WHERE price > \text{ALL} (SELECT price
    FROM Purchase
    WHERE maker='Gizmo-Works')
```
Correlated Queries

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

\[
\text{SELECT DISTINCT title} \\
\text{FROM Movie AS } x \\
\text{WHERE year} \neq \text{ANY} \\
(\text{SELECT year} \\
\text{FROM Movie} \\
\text{WHERE title} = x.\text{title});
\]

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

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

\[
\text{SELECT DISTINCT pname, maker} \\
\text{FROM Product AS } x \\
\text{WHERE price} \neq \text{ALL} (\text{SELECT price} \\
\text{FROM Product AS } y \\
\text{WHERE x.maker} = y.maker \text{ AND } y.year < 1972);
\]

Powerful, but much harder to optimize!
Existential/Universal Conditions

Product (pname, price, company)
Company(cname, city)

Find all companies s.t. some of their products have price < 100

```
SELECT DISTINCT Company.cname
FROM   Company, Product
WHERE  Company.cname = Product.company and Product.price < 100
```

Existential: easy ! 😊

Existential/Universal Conditions

Product (pname, price, company)
Company(cname, city)

Find all companies s.t. all of their products have price < 100

Universal: hard ! 😞
Existential/Universal Conditions

1. Find the other companies: i.e. s.t. some product ≥ 100

```sql
SELECT DISTINCT Company.cname
FROM Company
WHERE Company.cname IN (SELECT Product.company
                          FROM Product
                          WHERE Product.price >= 100)
```

2. Find all companies s.t. all their products have price < 100

```sql
SELECT DISTINCT Company.cname
FROM Company
WHERE Company.cname NOT IN (SELECT Product.company
                            FROM Product
                            WHERE Product.price >= 100)
```

Aggregation

```sql
SELECT Avg(price)
FROM Product
WHERE maker="Toyota"
```

SQL supports several aggregation operations:

SUM, MIN, MAX, AVG, COUNT
Aggregation: Count

Except COUNT, all aggregations apply to a single attribute.

COUNT applies to duplicates, unless otherwise stated:

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

Better:

```
SELECT Count(DISTINCT category)
FROM   Product
WHERE  year > 1995
```
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'
```

<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.

Purchase(product, date, price, quantity)

Example 2: find total sales after 9/1 per product.

```sql
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > "9/1"
GROUP BY product
```

Let’s see what this means…

---

Grouping and Aggregation

1. Compute the **FROM** and **WHERE** clauses.
2. Group by the attributes in the **GROUP BY**
3. Produce one tuple for every group by applying aggregation

**SELECT** can have (1) grouped attributes or (2) aggregates.
First compute the **FROM-WHERE** clauses (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>

```sql
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > "9/1"
GROUP BY product
```
GROUP BY v.s. Nested Quereis

```sql
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > "9/1"
GROUP BY product
```

```sql
SELECT DISTINCT x.product, (SELECT Sum(y.price*y.quantity)
FROM Purchase y
WHERE x.product = y.product
AND y.date > '9/1')
AS TotalSales
FROM Purchase x
WHERE x.date > "9/1"
```

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?

```sql
SELECT product, Sum(price * quantity) AS SumSales
       Max(quantity) AS MaxQuantity
FROM Purchase
GROUP BY product
```
HAVING Clause

Same query, except that we consider only products that had at least 30 items sold.

```
SELECT  product, Sum(price * quantity)
FROM    Purchase
WHERE   date > "9/1"
GROUP BY product
HAVING  Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.

General form of Grouping and Aggregation

```
SELECT  S
FROM    R_1, ..., R_n
WHERE   C_1
GROUP BY a_1, ..., a_k
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, ..., R_n
C_2 = is any condition on aggregate expressions
General form of Grouping and Aggregation

SELECT S
FROM R_1, ..., R_n
WHERE C_1
GROUP BY a_1, ..., a_k
HAVING C_2

Evaluation steps:
1. Compute the FROM-WHERE part, obtain a table with all attributes in R_1, ..., R_n
2. Group by the attributes a_1, ..., 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)
Grouping vs. Nested Queries

• Find all authors who wrote at least 10 documents:

• Attempt 1: with nested queries

```
SELECT DISTINCT Author.name
FROM Author
WHERE count(SELECT Wrote.url
            FROM Wrote
            WHERE Author.login=Wrote.login)
    > 10
```

This is SQL by a novice

• Attempt 2: SQL style (with GROUP BY)

```
SELECT Author.name
FROM Author, Wrote
WHERE Author.login=Wrote.login
GROUP BY Author.name
HAVING count(wrote.url) > 10
```

No need for DISTINCT: automatically from GROUP BY
Grouping vs. Nested Queries

• Find all authors who have a vocabulary over 10000 words:

```
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
```

Look carefully at the last two queries: you may be tempted to write them as a nested queries, but in SQL we write them best with GROUP BY

NULLS in SQL

• Whenever we don’t have a value, we can put a NULL
• Can mean many things:
  – Value does not exists
  – Value exists but is unknown
  – Value not applicable
  – Etc.
• The schema specifies for each attribute if it can be null (nullable attribute) or not
• How does SQL cope with tables that have NULLs?
Null Values

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

• If x = NULL then x = “Joe” is UNKNOWN

• In SQL there are three boolean values:
  FALSE = 0
  UNKNOWN = 0.5
  TRUE = 1

Null Values

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

E.g.
age=20
height=NULL
weight=200

Rule in SQL: include only tuples that yield TRUE
Null Values

Unexpected behavior:

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

Some Persons are not included!

Null Values

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

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

Now it includes all Persons
Outerjoins

Explicit joins in SQL:

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

\[
\begin{align*}
\text{SELECT} & \quad \text{Product.name, Purchase.store} \\
\text{FROM} & \quad \text{Product JOIN Purchase ON} \\
& \quad \text{Product.name = Purchase.prodName}
\end{align*}
\]

Same as:

\[
\begin{align*}
\text{SELECT} & \quad \text{Product.name, Purchase.store} \\
\text{FROM} & \quad \text{Product, Purchase} \\
\text{WHERE} & \quad \text{Product.name = Purchase.prodName}
\end{align*}
\]

But Products that never sold will be lost!

Outerjoins

Left outer joins in SQL:

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

\[
\begin{align*}
\text{SELECT} & \quad \text{Product.name, Purchase.store} \\
\text{FROM} & \quad \text{Product LEFT OUTER JOIN Purchase ON} \\
& \quad \text{Product.name = Purchase.prodName}
\end{align*}
\]
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
Modifying the Database

Three kinds of modifications
• Insertions
• Deletions
• Updates

Sometimes they are all called “updates”

Insertions

General form:

```
INSERT INTO R(A1, ..., An) VALUES (v1, ..., vn)
```

Example: Insert a new purchase to the database:

```
INSERT INTO Purchase(buyer, seller, product, store)
VALUES (‘Joe’, ‘Fred’, ‘wakeup-clock-espresso-machine’,
        ‘The Sharper Image’)
```

Missing attribute → NULL.
May drop attribute names if give them in order.
Insertions

```sql
INSERT INTO PRODUCT(name)
SELECT DISTINCT Purchase.product
FROM Purchase
WHERE Purchase.date > "10/26/01"
```

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

---

Insertion: an Example

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

prodName is foreign key in Product.name

Suppose database got corrupted and we need to fix it:

<table>
<thead>
<tr>
<th>Product</th>
<th>Purchase</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>buyerName</td>
</tr>
<tr>
<td>gizmo</td>
<td>Smith</td>
</tr>
<tr>
<td></td>
<td>Smith</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>prodName</th>
<th>buyerName</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>camera</td>
<td>John</td>
<td>200</td>
</tr>
<tr>
<td>gizmo</td>
<td></td>
<td>80</td>
</tr>
</tbody>
</table>

Task: insert in Product all prodNames from Purchase.
**Insertion: an Example**

```
INSERT INTO Product(name)
SELECT DISTINCT prodName
FROM Purchase
WHERE prodName NOT IN (SELECT name FROM Product)
```

<table>
<thead>
<tr>
<th>name</th>
<th>listPrice</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>gizmo</td>
<td>100</td>
<td>Gadgets</td>
</tr>
<tr>
<td>camera</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Insertion: an Example**

```
INSERT INTO Product(name, listPrice)
SELECT DISTINCT prodName, price
FROM Purchase
WHERE prodName NOT IN (SELECT name FROM Product)
```

<table>
<thead>
<tr>
<th>name</th>
<th>listPrice</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>gizmo</td>
<td>100</td>
<td>Gadgets</td>
</tr>
<tr>
<td>camera</td>
<td>200</td>
<td>-</td>
</tr>
<tr>
<td>camera ?</td>
<td>225</td>
<td>-</td>
</tr>
</tbody>
</table>
Deletions

Example:

```sql
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:

```sql
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 see 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

- Characters:
  - CHAR(20) -- fixed length
  - VARCHAR(40) -- variable length

- Numbers:
  - INT, REAL plus variations

- Times and dates:
  - DATE, TIME (Pointbase)
Creating Tables

Example:

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

Deleting or Modifying a Table

Deleting:

Example: `DROP Person;`

Exercise with care !!

Altering: (adding or removing an attribute).

Example:

```sql
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:

```
CREATE TABLE Person(
  name VARCHAR(30),
  social-security-number INT,
  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)

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

Sequential scan of the file Person may take long
Indexes

• Create an index on name:

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

• B+ trees have fan-out of 100s: max 4 levels!
Creating Indexes

Indexes can be useful in range queries too:

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

B+ trees help in:

```
SELECT *
FROM Person
WHERE age > 25 AND age < 28
```

Why not create indexes on everything?

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”
```

and even in:

```
SELECT *
FROM Person
WHERE age = 55
```

But not in:

```
SELECT *
FROM Person
WHERE city = “Seattle”
```
The Index Selection Problem

• We are given a workload = a set of SQL queries plus how often they run
• What indexes should we build to speed up the workload?
• FROM/WHERE clauses → favor an index
• INSERT/UPDATE clauses → discourage an index
• Index selection = normally done by people, recently done automatically

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
  - Pre-computed offline – fast at runtime
  - May have stale data
Updating Views

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

Employee(ssn, 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(ssn, name, department, project, salary)
VALUES(NULL, "Joe", NULL, "Optimizer", NULL)
```

Non-Updatable Views

Person(name, city)
Purchase(buyer, seller, product, store)

```
CREATE VIEW City-Store AS
SELECT Person.city, Purchase.store
FROM Person, Purchase
WHERE Person.name = Purchase.buyer
```

How can we add the following tuple to the view?

(“Seattle”, “Nine West”)

We don’t know the name of the person who made the purchase; cannot set to NULL (why?)
Summary

• Data Manipulation Language
  – SELECT, INSERT

• Data Definition Language
  – CREATE, DELETE, DROP