SQL: Nested Queries, Group By, Having

Prof. Hyuk-Yoon Kwon
https://sites.google.com/view/seoultech-bigdata

Today's Lecture

1. **Multiset operators & Nested queries**
2. **Aggregation & GROUP BY**
3. **Advanced SQL-izing**

Most parts are based on slides used in Stanford (http://web.stanford.edu/class/cs145)

INTERSECT: Still some subtle problems…

*"Headquarters of companies which make gizmos in US **AND** China"*

Company(name, hq_city)
Product(pname, maker, factory_loc)

```
SELECT hq_city
FROM   Company, Product
WHERE  maker = name
     AND factory_loc = 'US'
INTERSECT
SELECT hq_city
FROM   Company, Product
WHERE  maker = name
          AND factory_loc = 'China'
```

Company(name, hq_city) AS C
Product(pname, maker, factory_loc) AS P

```
SELECT hq_city
FROM   Company, Product
WHERE  maker = name
AND    factory_loc='US'
INTERSECT
SELECT hq_city
FROM   Company, Product
WHERE maker = name
AND factory_loc='China'
```

Example : C JOIN P on maker = name

C.name	C.hq_city	P.pname	P.maker	P.factory_loc
X Co.	Seattle	X	X Co.	U.S.
Y Inc.	Seattle	X	Y Inc.	China

X Co has a factory in the US (but not China)
Y Inc. has a factory in China (but not US)

But Seattle is returned by the query!

WHERE 절에서 중복된 값을 가진 행 제거 X -> 중복된 값 그대로 반환

One Solution: Nested Queries

Company(name, hq_city)
Product(pname, maker, factory_loc)

```
SELECT DISTINCT hq_city
FROM   Company, Product
WHERE  maker = name
     AND name IN (
                  SELECT maker
                  FROM   Product
                  WHERE  factory_loc = 'US')
          AND name IN (
                  SELECT maker
                  FROM   Product
                  WHERE  factory_loc = 'China')
```

IN {a, b, c} equals to
(a = a) or (a = b) or (a = c)

IN : WHERE 절에서 사용
-> 특정 column 값이 주어진 목록 안에 있는지 여부 확인
-> Sub-Query와 함께 사용 가능
(Sub-Query 결과 : 값 목록으로 반환)

High-level note on nested queries

- **We can do nested queries because SQL is *compositional:***
  - Everything (inputs / outputs) is represented as multisets-
  - the output of one query can thus be used as the input to another (nesting)!
- **This is extremely powerful!**

Compositional, 구성 가능성
: SQL에서 모든 입출력이 다중 집합으로 표현됨
-> " 한 Query의 출력 = 다른 Query 입력 " 가능
=> Nested (중첩) Query 가능

Nested Queries : Sub-queries Return Relations

Another example

```
Company(name, city)
Product(name, maker)
Purchase(id, product, buyer)
```

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

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

Sub-Query (중첩)
: 다른 Query의 WHERE 절 내에 포함된 Query
-> Sub-Query Result = Relation (집합)

"Cities where one can find companies that manufacture products bought by Joe Blow"

Nested Queries Return Relations

You can also use operations of the form:

- s > ALL R
- s < ANY R
- EXISTS R

ALL : Sub-Query의 모든 반환 값이 주 Query에서 비교 연산자와 일치해야 함
ex) all value >= 5 : All Sub-Query return value >=5

ANY : Sub-Query 반환 값 중 하나라도 주 쿼리 비교 연산자와 일치하면 됨

EXISTS : Sub-Query 반환 결과 존재 시 true 반환

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

Product(name, price, category, maker)

```
SELECT name
FROM   Product
WHERE  price > ALL(
          SELECT  price
          FROM    Product
          WHERE maker = 'Gizmo-Works')
```

Product(name, price, category, maker)

```
SELECT p1.name
FROM   Product p1
WHERE  p1.maker = 'Gizmo-Works'
  AND EXISTS(
          SELECT   p2.name
          FROM     Product p2
          WHERE  p2.maker <> 'Gizmo-Works'
            AND p1.name = p2.name)
```

Find 'copycat' products,
i.e. products made by competitors with the same names as products made by "Gizmo-Works"

<> means !=

Nested queries as alternatives to INTERSECT and MINUS

```
(SELECT R.A, R.B
FROM   R)
INTERSECT
(SELECT S.A, S.B
FROM   S)
```

```
SELECT R.A, R.B
FROM   R
WHERE EXISTS(
          SELECT *
          FROM S
          WHERE R.A=S.A AND R.B=S.B)
```

```
(SELECT R.A, R.B
FROM   R)
MINUS
(SELECT S.A, S.B
 FROM   S)
```

```
SELECT R.A, R.B
FROM   R
WHERE NOT EXISTS(
          SELECT *
          FROM S
          WHERE R.A=S.A AND R.B=S.B)
```

Correlated Queries Using External Vars in Nested Queries

Find movies whose title appears more than once.

Movie(title, year, director, length)

```
SELECT DISTINCT title
FROM Movie AS m
WHERE year <> ANY(
        SELECT year
        FROM Movie
        WHERE title = m.title)
```

Correlated Query, 연관 서브 쿼리

: 외부 Query와 관련된 결과를 내기 위해 Sub-Query에 외부 Query 변수 사용

-> 외부 Query 각 행 마다 내부 Query 1회 실행

-> 외부 변수 값에 따라 연관 Sub-Query 결과 변화

(일반적인 Sub-Query는 외부 Query와 독립적 실행)

-> 여러 Table에서 data 가져와서 비교 가능

-> But 실행 속도 느림 & 복잡한 Query 작성 시 가독성 떨어짐

Complex Correlated Query Example)

Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972.

Product(name, price, category, maker, year)

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

Basic SQL Summary

- **SQL provides a high-level declarative language for manipulating data (DML)**
- **The main structure is the SFW block**
- **Powerful, nested queries also allowed.**

2. Aggregation & GROUP BY

What you will learn about in this section

1. Aggregation operators
2. GROUP BY
3. GROUP BY: with HAVING, semantics

Aggregation

SQL supports several **aggregation** operations: SUM, COUNT, MIN, MAX, AVG

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

```
SELECT COUNT(*)
FROM   Product  W
HERE     year > 1995
```

Except COUNT, all aggregations apply to a single attribute

Aggregation, 집계

: Data 집계하여 요약 정보 생성하는 것

-> 일반적으로 GROUP BY 절과 함께 사용

-> 일반적으로 SELECT문의 SELECT / HAVING 절에서 사용

-> SELECT 절에서 : 집계 함수와 함께 grouping 할 열 나열

-> HAVING 절에서 : 집계 함수에 대한 조건 지정

=> 대규모 data 요약하여 정보 파악 시 유용

COUNT applies to duplicates, unless otherwise stated

```
SELECT COUNT(category)
FROM   Product
WHERE  year > 1995
```

```
SELECT COUNT(DISTINCT category)
FROM   Product  W
HERE     year > 1995
```

Note: Same as COUNT(*), Why?

COUNT(*) counts all the rows returned by a query, including duplicates

-> DISTINCT 사용으로 중복 제거 후 개수 세기 가능

More Examples

```
Purchase(product, date, price, quantity)
```

```
SELECT SUM(price * quantity)
FROM   Purchase
```

```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = 'bagel'
```

What do these mean?

Grouping and Aggregation - Semantics of the query:

Purchase(product, date, price, quantity)

```
SELECT   product,
         SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Find total sales after 10/1/2005 per product.

GROUP BY : Data 그룹화에 사용되는 기능
-> SELECT 문에 사용됨
-> 결과를 지정된 COLUMN의 고유 값으로 그룹화
=> Group별 집계 수행 가능

1. Compute the FROM and WHERE clauses

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

FROM

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

2. Group by the attributes in the GROUP BY

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

GROUP BY

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

3. Compute the SELECT clause: grouped attributes and aggregates

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

SELECT

Product	TotalSales
Bagel	50
Banana	15

General form of Grouping and Aggregation

```
SELECT    S
FROM      R1,…,Rn
WHERE     C1 GRO
UP BY       a1,…,ak
HAVING    C2
```

- **S = Can ONLY contain attributes $a_1,\ldots,a_k$ and/or aggregates over other attributes**
- **C_1 = is any condition on the attributes in $R_1,\ldots,R_n$**
- **C_2 = is any condition on the aggregate expressions**

S = 그룹화된 결과를 나타내는 Attributes / 다른 Attributes에 대한 집계 결과 포함
C1 = R 속성 조건 (WHERE 절과 유사하며 그룹화 전에 Filtering 됨)
C2 = 집계 표현식 조건 (HAVING 절에서 정의되며 그룹화된 결과에 대한 Filtering 수행)

Evaluation steps:

1. Evaluate FROM-WHERE: apply condition C_1 on the attributes in $R_1,\ldots,R_n$
2. GROUP BY the attributes $a_1,\ldots, a_k$
3. Apply condition C_2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

HAVING Clause

Same query as before, except that we consider only products that have more than 100 buyers.

```
SELECT   product, SUM(price*quantity)
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
HAVING   SUM(quantity) > 100
```

HAVING : "그룹화된 결과"에 대한 조건 지정 용도

-> GROUP BY 절 다음에 조건 위치 (함께 사용)

-> WHERE 절과 유사하게 작동하나 집계 함수와 함께 사용됨

(WHERE 절은 DB Table의 모든 개별 tuple에 대한 조건 지정)

HAVING clauses contains conditions on **aggregates** *Whereas WHERE clauses condition on **individual tuples…***

Group-by v.s. Nested Query

- **Find authors who wrote at least 10 documents:**

```
Author(login, name)
Wrote(login, url)
```

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

- **Attempt 1: with nested queries**

  -> How many times do we do a SFW query over all the Wrote relations?

- **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
```

  -> How about when written this way?

With GROUP BY can be **much** more efficient!

3. Advanced SQL-izing

What you will learn about in this section

1. **Quantifiers**
2. **NULLs**
3. **Outer Joins**

Quantifiers

Product(name, price, company)
Company(name, city)

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

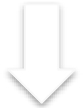
Find all companies that make some products with price < 100

Quantifiers, 양자
: 조건식에서 사용하여 해당 조건식의 참 / 거짓 여부 결정
ALL : 모든 값이 만족해야 TRUE
ANY / SOME : 하나 이상 만족하는 값 있으면 TRUE
EXISTS : 만족하는 행 존재 시 TRUE
IN : 지정된 값 중 하나라도 만족 시 TRUE

An **existential quantifier** is a logical quantifier (roughly) of the form "there exists"

Existential Quantifier, 존재 양자 (논리적 양자의 한 종류)
: 조건식 만족하는 data 존재 시 true 반환
-> WHERE EXISTS 같은 형태로 쓰임
⇒ 해당 조건 만족하는 data 존재하는지 확인하는 역할 (Easy)

Find all companies with products all having price < 100



Equivalent

A **universal quantifier** is of the form "for all"

Universal Quantifier, 범위 양자 (논리적 양자의 한 종류)
: 조건식 만족하는 모든 data가 해당 조건에 부합하는지 확인
-> WHERE ALL 같은 형태로 쓰임
⇒ DB에서 조건식 만족하는 모든 data가 부합한가 판단하는 역할 (Hard)

Exclude that all companies that make some products with price >= 100

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

Null Values

- **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 …
- **The schema specifies for each attribute if can be null or not**
- **How does SQL cope with tables that have NULLs?**

NULL 값 가진 Table 처리 방식

1) NULL 값 비교

: 비교 연산자 통해 비교 가능 => IS NULL / NOT NULL 연산자 사용

2) NULL 값 처리

: 다른 값과 연산 시 NULL 반환 (결과와 예상 다를 수 있음)

=> IFNULL, COALESCE … 등의 함수 사용

3) NULL 값 처리 위한 제약

: DB Schema에서 Attribute에 NULL 값 허용하지 않도록 제약 조건 설정 가능

```
SELECT *
FROM   Person  W
HERE (age < 25)
  AND (height > 6 AND weight > 190)
```

A tuple (age=20 height=NULL weight=200) ?

Rule in SQL: include only tuples that yield TRUE (1)

조건식 결과 참인 data만을 검색하여 결과에 포함한다.

- ***For numerical operations,* NULL -> NULL:**
  - If x = NULL then 4*(3-x)/7 is still NULL
- ***For boolean operations,* in SQL there are three values:**

FALSE	**= 0**	**UNKNOWN**	**= 0.5**
TRUE	**= 1**		

Unexpected behavior:

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

Now it includes all Persons!

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

RECAP: Inner Joins

Inner Join : 2개 이상의 table에서 특정 조건 만족하는 행(row) 결합하여 새로운 Table 생성하는 Join 중 하나
-> 두 Table에서 일치하는 값만 연결 & 일치하지 않는 값 제외됨
= NULL 값 포함된 data 제외됨
(NULL 값 가진 record는 다른 table과 불일치하기 때문)
-> Cross join과 유사하게 작동
-> 특히 table 간 relationship 존재 시 자주 쓰임

By default, joins in SQL are "inner joins":

```
Product(name, category)  P
urchase(prodName, store)
```

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

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

Both equivalent:
Both INNER JOINS!

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

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

Note: another equivalent way to write an INNER JOIN!

name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Inner Joins + NULLS = Lost data?

However: Products that never sold (with no Purchase tuple) will be lost!

Inner Join : 두 Table 간에 일치하는 Record만 반환
-> 판매 X로 Purchase Table에 해당 제품에 대한 Tuple이 없다면 해당 제품은 결과에서 제외 = data 손실 발생
⇒ Data Modeling 시 해당 경우 고려하여 손실 방지 가능한 Join 유형 선택 필요 Ex) Outer join

Outer Joins

Outer Join
: 2개 이상 Table에서 조건에 따라 매칭되지 않는 값 포함하여 결합한 결과 출력하는 Join 중 하나
= 일치하지 않는 값 포함하여 모든 행 출력

- **An outer join returns tuples from the joined relations that don't have a corresponding tuple in the other relations**
  - i.e., If we join relations A and B on a.X = b.X, and there is an entry in A with X=5, but none in B with X=5…
    - A LEFT OUTER JOIN will return a tuple (a, NULL)!

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

1) 왼쪽 Table의 모든 Record와 일치하는 오른쪽 Table의 Record 반환
2) 불일치 시 NULL 값 반환

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

1) 오른쪽 Table의 모든 Record와 일치하는 왼쪽 Table의 Record 반환
2) 불일치 시 NULL 값 반환

- **Full outer join:**
  - Include the both left and right tuples even if there's no match

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

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

Now we'll get products even if they didn't sell

=> Left Outer Join + Right Outer Join 결과 합친 것과 동일
= 왼쪽 & 오른쪽 Table의 모든 Record 포함
1) 일치하는 경우 해당 값 반환
2) 불일치 시 NULL 값 반환