CS 6400 A

Database Systems Concepts and Design

Lecture 3

08/31/26

Logistics

Assignment 1 released today

- due Sep 16 @ 11:59PM
- You are NOT allowed to use AI for this assignment.

OH starting this week:

- Instructor (KACB 3322): Fridays 10-10:50AM
- TAs (common area near KACB 3319) :
  - Tuesday, Wednesday, Thursday 2-3PM

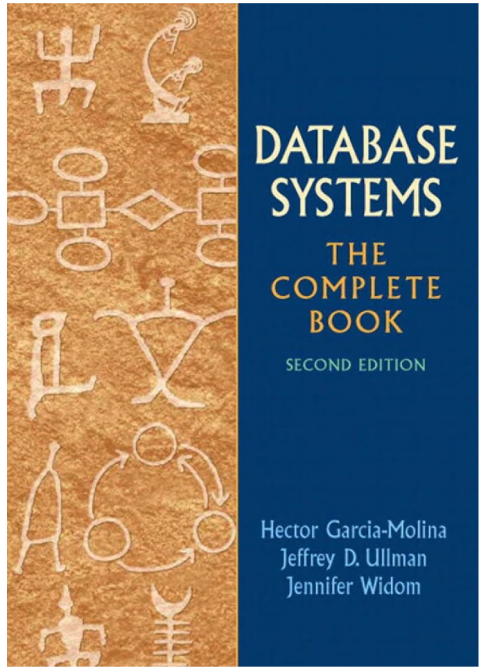
Agenda

1. Set operators & nested queries
2. Aggregation & GROUP BY
3. Advanced SQL-izing

Reading Materials

Database Systems: The Complete Book (2nd edition)

- Chapter 6: The Database Language SQL (6.2-6.4)

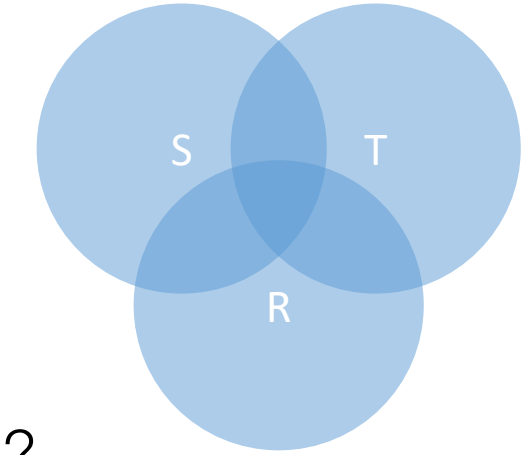


Acknowledgement: The following slides have been adapted from CS145 (Intro to Big Data Systems) taught by Peter Bailis.

1. Set Operators & Nested Queries

An Unintuitive Query

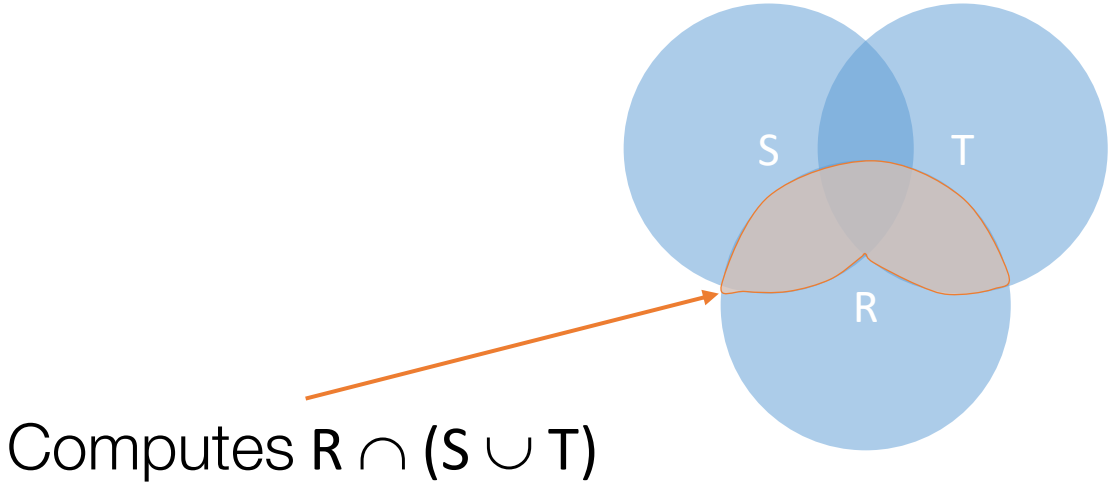
```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```



What does it compute?

An Unintuitive Query

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```



But what if S = ϕ?

Go back to the semantics!

An Unintuitive Query

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```

Recall the semantics!

1. Take cross-product
2. Apply selections / conditions
3. Apply projection

If S = {}, then the cross product of R, S, T = {}, and the query result = {}!

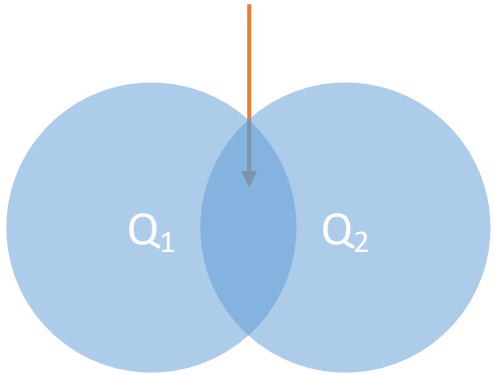
Must consider semantics here.
Are there more explicit way to do set operations like this?

Set Operations in SQL

Explicit Set Operators: INTERSECT

```
SELECT R.A
FROM   R, S
WHERE  R.A=S.A
INTERSECT
SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```

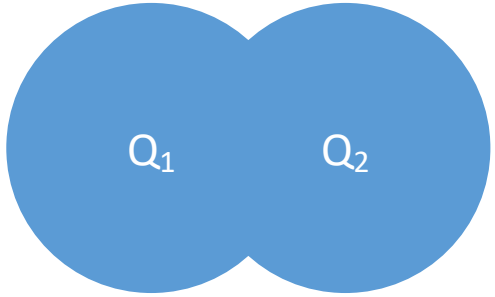
$$\{r.A \mid r.A = s.A\} \cap \{r.A \mid r.A = t.A\}$$



UNION

```
SELECT  R.A
FROM    R, S
WHERE   R.A=S.A
UNION
SELECT R.A
FROM    R, T
WHERE   R.A=T.A
```

$$\{r.A \mid r.A = s.A\} \cup \{r.A \mid r.A = t.A\}$$



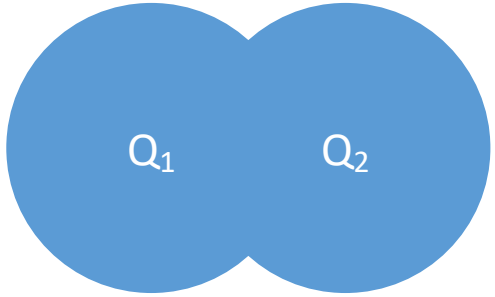
By default:
SQL uses set semantics!

What if we want duplicates?

UNION

```
SELECT  R.A
FROM    R, S
WHERE   R.A=S.A
UNION
SELECT R.A
FROM    R, T
WHERE   R.A=T.A
```

$$\{r.A \mid r.A = s.A\} \cup \{r.A \mid r.A = t.A\}$$



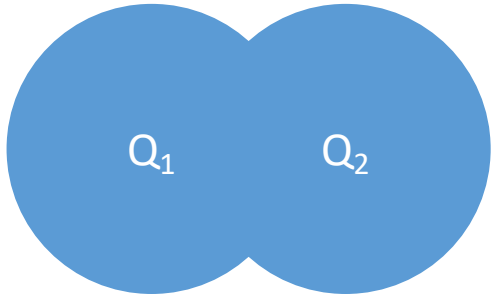
By default:
SQL uses set semantics!

What if we want duplicates?

UNION ALL

```
SELECT  R.A
FROM    R, S
WHERE   R.A=S.A
UNION ALL
SELECT R.A
FROM    R, T
WHERE   R.A=T.A
```

$$\{r.A \mid r.A = s.A\} \cup \{r.A \mid r.A = t.A\}$$

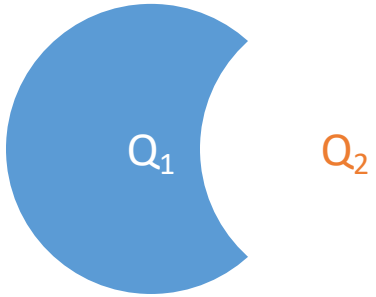


ALL indicates Multiset operations

EXCEPT

```
SELECT R.A
FROM   R, S
WHERE  R.A=S.A
EXCEPT
SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```

$$\{r.A \mid r.A = s.A\} \backslash \{r.A | r.A = t.A\}$$



Union Compatibility

```
Student(name VARCHAR, age INT, major VARCHAR)
Employee(ename VARCHAR, age INT, salary FLOAT)
```

```
SELECT  name
FROM    Student
UNION
SELECT ename
FROM    Employee
```

OK

```
SELECT  name, age
FROM    Student
UNION
SELECT age
FROM    Employee
```

Invalid

```
SELECT  name
FROM    Student
UNION
SELECT age
FROM    Employee
```

Invalid

Set operations require compatible schemas

INTERSECT: Still some subtle problems…

```
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'
```

"Headquarters of companies which make products in US AND China"

What can go wrong here?

INTERSECT: Remember the semantics!

```
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!

One Solution: **Nested Queries**

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

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

"Headquarters of companies which make products in US AND China"

One Solution: Nested Queries

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

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

"Headquarters of companies which make products in US AND China"

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!

Nested queries

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')
```

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

Nested Queries

Are these queries equivalent?

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

Beware of duplicates!

Nested Queries

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

```
SELECT DISTINCT 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')
```

Now they are equivalent (both use set semantics)

Subqueries Return Relations

You can also use operations of the form:

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

Ex: `Product(name, price, category, maker)`

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

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

Subqueries Return Relations

You can also use operations of the form:

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

Exists: not empty => true

Ex: `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)
```

<> means !=

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

Correlated Queries Using External Vars in Internal Subquery

```
Movie(title, year, director, length)
```

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

Find movies whose title appears more than once.

Note the scoping of the variables!

Subquery that refers to columns from the outer query.

Complex Correlated Query

```
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)
```

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

Can be very powerful (also much harder to optimize)

Correlated vs Regular Subqueries

In terms of execution

- Regular: executed once for the entire outer query
- Correlated: executed once **for each row processed** by the outer query (due to the dependence between inner and outer queries)

This means that correlated subqueries are usually very slow

- Many correlated subqueries can be rewritten as joins; modern optimizers often perform this transformation automatically.

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

```
SELECT DISTINCT m1.title
FROM Movie m1 JOIN Movie m2
       ON m1.title = m2.title
WHERE m1.year <> m2.year
```

Basic SQL Summary

- SQL provides a high-level declarative language for manipulating data (DML)
- The workhorse is the SFW block
- Set operators are powerful but have some subtleties
- Powerful, nested queries also allowed.

2. Aggregation & GROUP BY

Aggregation

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

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

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

Except COUNT, all aggregations apply to a single attribute

Aggregation: COUNT

COUNT applies to duplicates, unless otherwise stated

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

We probably want:

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

More Examples

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

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

What do these mean?

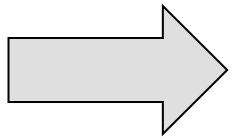
```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = 'bagel'
```

Simple Aggregations

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

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



50 (= 1*20 + 1.50*20)

Grouping and Aggregation

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

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

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

Let's see what this means…

Grouping and Aggregation

Semantics of the query:

1. Compute the FROM and WHERE clauses

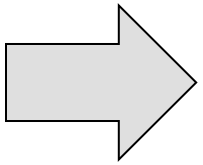
2. Group by the attributes in the GROUP BY

3. Compute the SELECT clause: grouped attributes and aggregates

1. Compute the FROM and WHERE clauses

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

FROM



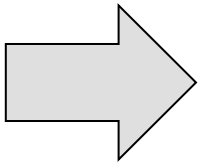
Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

2. Group by the attributes in the GROUP BY

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



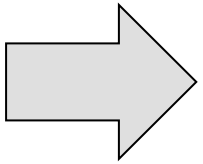
Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

3. Compute the SELECT clause: grouped attributes and aggregates

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

HAVING Clause

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

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

HAVING clauses contains **conditions on aggregates**

Whereas WHERE clauses condition on individual tuples…

General form of Grouping and Aggregation

```
SELECT      S
FROM        R1,…,Rn
WHERE       C1
GROUP 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

General form of Grouping and Aggregation

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

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

Group-by v.s. Nested Query

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

Find authors who wrote >= 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

Group-by v.s. Nested Query

Find all authors who wrote at least 10 documents:

- 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
```

This is SQL by an expert

No need for DISTINCT: automatically from GROUP BY

Group-by vs. Nested Query

Which way is more efficient?

- Attempt #1- With nested: How many times do we do a SFW query over all of the Wrote relations?

- Attempt #2- With group-by: How about when written this way?

With GROUP BY can be much more efficient!

3. Advanced SQL-izing

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 can be null (nullable attribute) or not

- How does SQL cope with tables that have NULLs?

Null Values

- 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

Comparisons with NULL result in UNKNOWN

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

Null Values

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

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

Won't return e.g. (age=20 height=NULL weight=200)!

Rule in SQL: include only tuples that yield TRUE (1.0)

Null Values

Unexpected behavior:

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

Does this query include all rows in the table?

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!

Inner Joins

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

```
Product(name, category)
Purchase(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!

Inner Joins + NULLS = Lost data?

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

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

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

Outer Joins

- 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 joins in SQL:

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

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

INNER JOIN:

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
```



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

LEFT OUTER JOIN:

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
  LEFT OUTER JOIN Purchase
       ON Product.name = Purchase.prodName
```



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

Other 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

Virtual views

- Relations defined by a query over other relations
- Do NOT exist physically
- Can be queried as if it were a stored table

```
CREATE VIEW GhibliMovies AS
  SELECT title, year, studioName
  FROM Movies
  WHERE studioName = 'Ghibli';
```

```
SELECT title
FROM GhibliMovies
WHERE year = 2008;
```

```
DROP VIEW GhibliMovies;
```

Materialized view

- A frequently-used view can be materialized (query results are stored)
- This makes reads faster; however, there is now a maintenance cost where view need to be updated whenever a base table changes

```
CREATE MATERIALIZED VIEW GhibliMovies AS
  SELECT title, year, studioName
  FROM Movies
  WHERE studioName = ‘Ghibli’;
```

When are views useful?

Advantages

- Security: expose only authorized rows/columns
- Query simplicity: hide complex queries behind a simple table-like interface
- Logical data independence: application based on views is independent of database

Disadvantages

- Performance: virtual views may require additional query processing over base tables
- Maintenance: materialized views must be refreshed when base data changes
- Update restrictions: updates only possible in limited circumstances

Summary

SQL is a rich programming language that handles the way data is processed declaratively