Lectures 2&3:
Introduction to SQL

Lecture 2: SQL Part I

Today's Lecture

1. SQL introduction & schema definitions
   - ACTIVITY: Table creation

2. Basic single-table queries
   - ACTIVITIY: Single-table queries!

3. Multi-table queries
   - ACTIVITY: Multi-table queries!

1. SQL Introduction & Definitions

What you will learn about in this section

1. What is SQL?

2. Basic schema definitions

3. Keys & constraints intro

4. ACTIVITY: CREATE TABLE statements

SQL Motivation

- Dark times 5 years ago.
  - Are databases dead?

- Now, as before: everyone sells SQL
  - Pig, Hive, Impala

- "Not-Yet-SQL?"

Basic SQL

SQL Introduction

- SQL is a standard language for querying and manipulating data

- SQL is a **very high-level** programming language
  - This works because it is optimized well!

> SQL stands for
> Structured Query Language

- Many standards out there:
  - ANSI SQL, SQL92 (a.k.a. SQL2), SQL99 (a.k.a. SQL3), ….
  - Vendors support various subsets

> *NB*: Probably the world's most successful **parallel** programming language (multicore?)

SQL is a…

- Data Definition Language (DDL)
  - Define relational *schemata*
  - Create/alter/delete tables and their attributes

- Data Manipulation Language (DML)
  - Insert/delete/modify tuples in tables
  - Query one or more tables – discussed next !

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

A **relation** or **table** is a multiset of tuples having the attributes specified by the schema

Let's break this definition down

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

A **multiset** is an unordered list (or: a set with multiple duplicate instances allowed)

List: [1, 1, 2, 3]
Set: {1, 2, 3}
Multiset: {1, 1, 2, 3}

I.e. no *next()*, etc. methods!

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

An **attribute** (or **column**) is a typed data entry present in each tuple in the relation

NB: Attributes must have an ***atomic*** *type in standard SQL, i.e. not a list, set, etc.*

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

*Also referred to sometimes as a **record***

A **tuple** or **row** is a single entry in the table having the attributes specified by the schema

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

The number of tuples is the **cardinality** of the relation

The number of attributes is the **arity** of the relation

14

Data Types in SQL

- Atomic types:
  - Characters: CHAR(20), VARCHAR(50)
  - Numbers: INT, BIGINT, SMALLINT, FLOAT
  - Others: MONEY, DATETIME, …

- Every attribute must have an atomic type   *Why?*
  - Hence tables are flat

15

Table Schemas

- The **schema** of a table is the table name, its attributes, and their types:

<div style="border:1px solid black">

Product(Pname: *string*, Price: *float*, Category: *string*, Manfacturer: *string*)

</div>

- A **key** is an attribute whose values are unique; we underline a key

<div style="border:1px solid black">

Product(<u>Pname</u>: *string*, Price: *float*, Category: *string*, <u>Manfacturer</u>: *string*)

</div>

16

Key constraints

A <u>key</u> is a **minimal subset of attributes** that acts as a unique identifier for tuples in a relation

- A key is an implicit constraint on which tuples can be in the relation

  - I.e. if two tuples agree on the values of the key, then they must be the same tuple!

```
Students(sid:string, name:string, gpa: float)
```

1. Which would you select as a key?
2. Is a key always guaranteed to exist?
3. Can we have more than one key?

NULL and NOT NULL

- To say "don't know the value" we use NULL
  - NULL has (sometimes painful) semantics, more detail later

```
Students(sid:string, name:string, gpa: float)
```

Sid	Name	GPA
123	Bob	3.9
143	Jim	NULL

Say, Jim just enrolled in his first class.

In SQL, we may constrain a column to be NOT NULL, e.g., name

General Constraints

- We can actually specify arbitrary assertions
  - E.g. *"There cannot be 25 people in the DB class"*

- In practice, we don't specify many such constraints. Why?
  - <u>Performance!</u>

> Whenever we do something ugly (or avoid doing something convenient) it's for the sake of performance

Summary of Schema Information

- Schema and Constraints are how databases understand the semantics (meaning) of data

- They are also useful for optimization

- SQL supports general constraints:
  - Keys and foreign keys are most important
  - We'll give you a chance to write the others.

ACTIVITY: Activity-2-1.ipynb

2. Single-table queries

What you will learn about in this section

1. The SFW query

2. Other useful operators: LIKE, DISTINCT, ORDER BY

3. ACTIVITY: Single-table queries

SQL Query

- Basic form (there are many many more bells and whistles)

```
SELECT <attributes>
FROM   <one or more relations>
WHERE  <conditions>
```
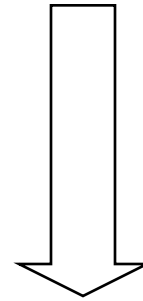
Call this a **SFW** query.

Simple SQL Query: Selection

Selection is the operation of filtering a relation's tuples on some condition

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT  *
FROM    Product
WHERE   category = 'Gadgets'
```
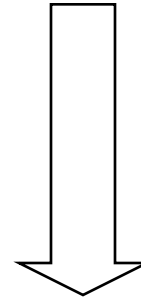
PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks

25

Simple SQL Query: Projection

Projection is the operation of producing an output table with tuples that have a subset of their prior attributes

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT Pname, Price, Manufacturer
FROM    Product
WHERE   category = 'Gadgets'
```
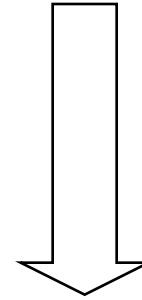
PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks

Notation

Input schema

Product(PName, Price, Category, Manfacturer)

```
SELECT Pname, Price, Manufacturer
FROM    Product
WHERE   category = 'Gadgets'
```

Output schema

Answer(PName, Price, Manfacturer)

A Few Details

- SQL **commands** are case insensitive:
  - Same: SELECT, Select, select
  - Same: Product, product

- **Values** are **not:**
  - <u>Different</u>: 'Seattle', 'seattle'

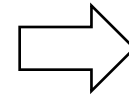- Use single quotes for constants:
  - 'abc' - yes
  - "abc" - no

LIKE: Simple String Pattern Matching

```
SELECT  *
FROM    Products
WHERE   PName LIKE '%gizmo%'
```

- s **LIKE** p: pattern matching on strings

- p may contain two special symbols:
  - % = any sequence of characters
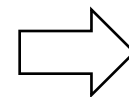 - _ = any single character

DISTINCT: Eliminating Duplicates

```
SELECT DISTINCT category
FROM    Product
```

⇨

Category
Gadgets
Photography
Household

Versus:

```
SELECT category
FROM    Product
```

⇨

Category
Gadgets
Gadgets
Photography
Household

30

ORDER BY: Sorting the Results

```
SELECT    pname, price, manufacturer
FROM      Product
WHERE     category='gizmo' AND price > 50
ORDER BY price, pname
```

Notes:
- Ties are broken by the second attribute on the ORDER BY list, etc.
- Ordering is ascending, unless you specify the DESC keyword.

ACTIVITY: Activity-2-2.ipynb

3. Multi-table queries

What you will learn about in this section

1. Foreign key constraints

2. Joins: basics

3. Joins: SQL semantics

4. ACTIVITY: Multi-table queries

34

Foreign Key constraints

- Suppose we have the following schema:

Students(<u>sid</u>: *string,* name: *string*, gpa: *float)*

Enrolled(<u>student_id</u>: *string,* <u>cid</u>: *string,* grade: *string)*

- And we want to impose the following constraint:
  - <u>'Only bona fide students may enroll in courses'</u> i.e. students that appear in the Students table

Students

Sid	Name	Gpa
101	Bob	3.2
123	Mary	3.8

Enrolled

studid	cid	Grade
123	564	A
123	537	A+

Student_id alone is not a key- what is?

We say that student_id is a **foreign key** that refers to Students

Declaring Foreign Keys

```
Students(sid: string, name: string, gpa: float)
Enrolled(student_id: string, cid: string, grade: string)

CREATE TABLE Enrolled(
      student_id  CHAR(20),
      cid         CHAR(20),
      grade       CHAR(10),
      PRIMARY KEY (student_id, cid),
      FOREIGN KEY (student_id) REFERENCES Students
)
```

Foreign Keys and update operations

Students(<u>sid</u>*: string,* name: *string*, gpa: *float*)

Enrolled(<u>student_id</u>: *string,* <u>cid</u>: *string*, grade: *string*)

- What if we insert a tuple into Enrolled, but no corresponding student?
  - INSERT is rejected (foreign keys are <u>constraints</u>)!

- What if we delete a student?

DBA chooses (syntax in the book)

1. Disallow the delete
2. Remove all of the courses for that student
3. *SQL allows a third via NULL (not yet covered)*

Keys and Foreign Keys

Company

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

What is a
foreign key vs.
a key here?

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

38

Joins

Product(<u>pname</u>, price, category, manufacturer)

Company(<u>cname</u>, stockPrice, country)

Note: we will often omit attribute types in schema definitions for brevity, but assume attributes are always types

Ex: Find all products under $200 manufactured in Japan; return their names and prices.

```
SELECT  PName, Price
FROM    Product, Company
WHERE   Manufacturer = CName
        AND Country='Japan'
        AND Price <= 200
```

39

Joins

Product(pname, price, category, manufacturer)

Company(cname, stockPrice, country)

Ex: Find all products under $200 manufactured in Japan; return their names and prices.

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
       AND Country='Japan'
       AND Price <= 200
```

A join between tables returns all unique combinations of their tuples which meet some specified join condition

40

Joins

Product(pname, price, category, manufacturer)

Company(cname, stockPrice, country)

Several equivalent ways to write a basic join in SQL:

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
        AND Country='Japan'
        AND Price <= 200
```

```
SELECT PName, Price
FROM    Product
JOIN Company ON Manufacturer = Cname
                        AND Country='Japan'

WHERE Price <= 200
```
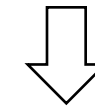
A few more later on…

Joins

Product

PName	Price	Category	Manuf
Gizmo	$19	Gadgets	GWorks
Powergizmo	$29	Gadgets	GWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

Company

Cname	Stock	Country
GWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT  PName, Price
FROM    Product, Company
WHERE   Manufacturer = CName
        AND Country='Japan'
        AND Price <= 200
```

PName	Price
SingleTouch	$149.99

42

Tuple Variable Ambiguity in Multi-Table

```
Person(name, address, worksfor)

Company(name, address)
```

```
SELECT DISTINCT name, address
FROM            Person, Company
WHERE           worksfor = name
```

Which "address" does this refer to?

Which "name"s??

Tuple Variable Ambiguity in Multi-Table

```
Person(name, address, worksfor)

Company(name, address)
```

Both equivalent ways to resolve variable ambiguity

```
SELECT DISTINCT   Person.name, Person.address
FROM              Person, Company
WHERE             Person.worksfor = Company.name
```

```
SELECT DISTINCT   p.name, p.address
FROM              Person p, Company c
WHERE             p.worksfor = c.name
```

Meaning (Semantics) of SQL Queries

```
SELECT  x₁.a₁, x₁.a₂, …, xₙ.aₖ
FROM    R₁ AS x₁, R₂ AS x₂, …, Rₙ AS xₙ
WHERE   Conditions(x₁,…, xₙ)
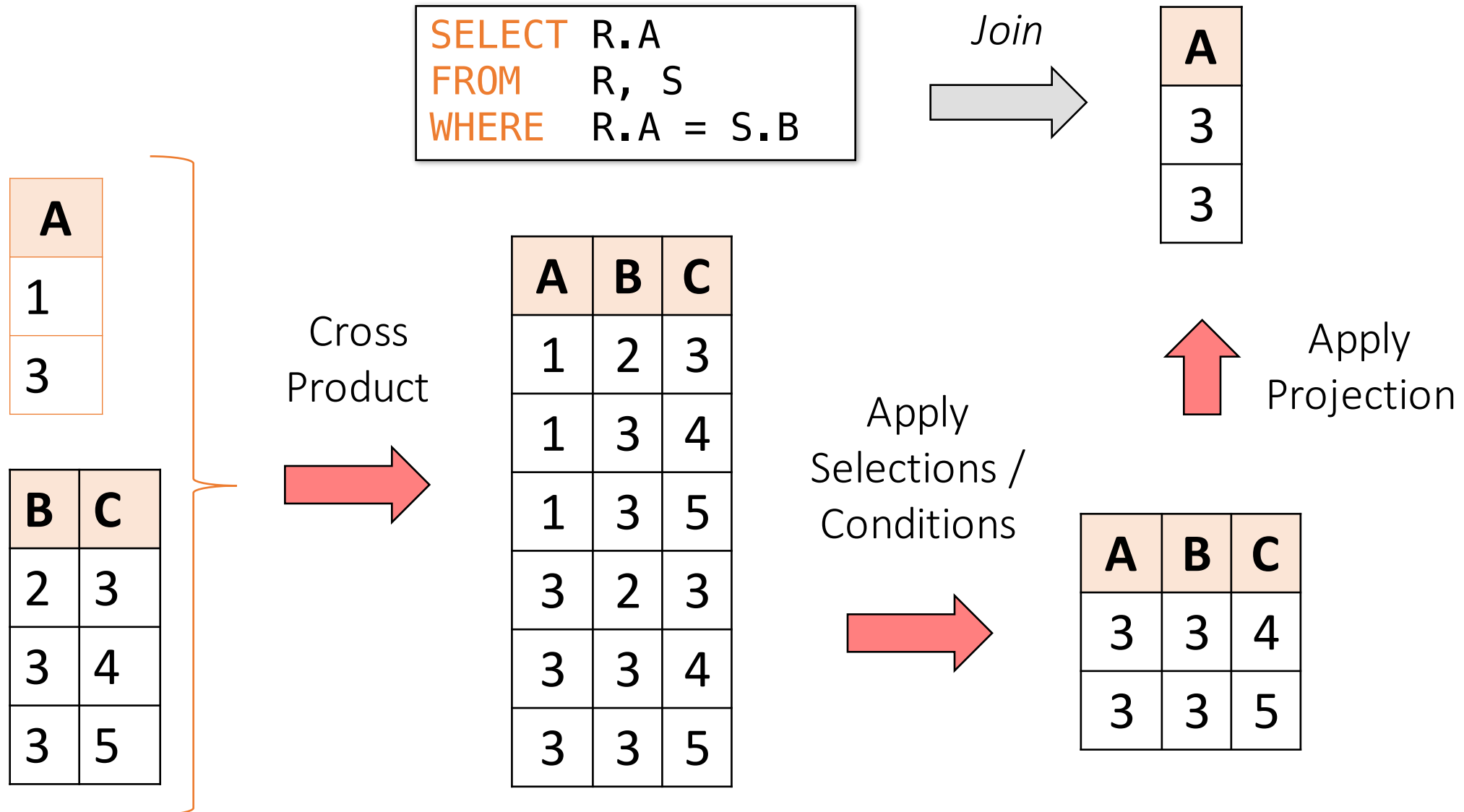```

Almost never the *fastest* way to do it!

Answer = {}
**for** $x_1$ **in** R**₁** **do**
    **for** $x_2$ **in** R**₂** **do**

        …..

            **for** $x_n$ **in** R$_n$ **do**

                **if** Conditions($x_1$,…, $x_n$)

                    **then** Answer = Answer $\cup$ {($x_1.a_1$, $x_1.a_2$, …, $x_n.a_k$)}

**return** Answer

**Note:** this is a *multiset* union

45

# An example of the semantics

```
SELECT  R.A
FROM    R, S
WHERE   R.A = S.B
```

Join

A
3
3

A
1
3

Cross Product

A	B	C
1	2	3
1	3	4
1	3	5
3	2	3
3	3	4
3	3	5

B	C
2	3
3	4
3	5

Apply Selections / Conditions

Apply Projection

A	B	C
3	3	4
3	3	5

46

Note the execution order of a join

```
SELECT  R.A
FROM    R, S
WHERE   R.A = S.B
```

1. Take **cross product**:
$$X = R \times S$$

Recall: Cross product (A X B) is the set of all unique tuples in A,B

Ex: {a,b,c} X {1,2}
     = {(a,1), (a,2), (b,1), (b,2), (c,1), (c,2)}

2. Apply **selections / conditions**:
$$Y = \{(r,s) \in X \mid r.A == r.B\}$$

= Filtering!

3. Apply **projections** to get final output:
$$Z = (y.A,) \ for \ y \in Y$$

= Returning only *some* attributes

Remembering this order is key to understanding the output of certain joins (see later on…)

47

A Subtlety about Joins

Product(<u>pname</u>, price, category, manufacturer)

Company(<u>cname</u>, stockPrice, country)

Find all countries that manufacture some product
in the 'Gadgets' category.

```
SELECT  Country
FROM    Product, Company
WHERE   Manufacturer=CName AND Category='Gadgets'
```
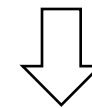
48

A subtlety about Joins

Product

PName	Price	Category	Manuf
Gizmo	$19	Gadgets	GWorks
Powergizmo	$29	Gadgets	GWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

Company

Cname	Stock	Country
GWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT Country
FROM   Product, Company
WHERE  Manufacturer=Cname
   AND Category='Gadgets'
```

Country
?
?

What is the problem ?
What's the solution ?

ACTIVITY: Lecture-2-3.ipynb

Lecture 3: SQL Part II

Today's Lecture

1. Set operators & nested queries
   - ACTIVITY: Set operator subtleties

2. Aggregation & GROUP BY
   - ACTIVITIY: Fancy SQL Part I

3. Advanced SQL-izing
   - ACTIVITY: Fancy SQL Part II

1. Set Operators & Nested Queries

What you will learn about in this section

1. ORDER BY semantics (cont'd)

2. Set operators

3. Nested queries

4. ACTIVITY: Set operator subtleties

Ordering

```
SELECT    Name
FROM      Product
ORDER BY Price
```

SQL-89 says "This makes no sense!"

- **Formally, the ordering should only be applied on the values returned**
  - Order of operations: SELECT FROM -> ORDER BY

- Intuitively though, clear what the above query means:
  - "Give me the products in descending order…"
  - Some DBMSs will allow you to do this

Ordering

```
SELECT    DISTINCT Name
FROM      Product
ORDER BY  Price
```

SQL-89 says "This <u>definitely</u> makes no sense!"

- **Formally, the ordering should only be applied on the values returned**
  - Order of operations: SELECT FROM -> ORDER BY

- Is the meaning of this one intuitively clear??
  - What if two products (from different manufacturers) have the same name, and different prices?
  - Some DBMSs allow you to do this still- how?

56

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
```


Computes R ∩ (S ∪ T)
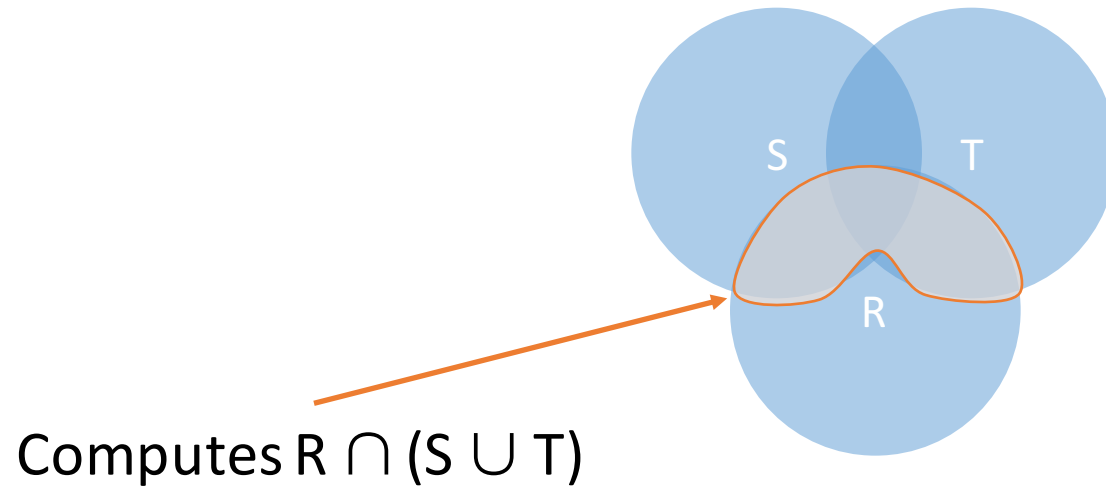
But what if S = φ?

What is the evaluation order here??

An Unintuitive Query

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

- Recall execution order of **joins!**
    1. Take cross-product
    2. Apply selections / conditions
    3. Apply projection

What? This query is a join? Is every multi-relation query a join...?? (Hint: <u>yes</u>)

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

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

59

INTERSECT

```
SELECT   R.A
FROM     R, S          ⎤ Q₁
WHERE    R.A=S.A       ⎦
INTERSECT
SELECT   R.A
FROM     R, T          ⎤ Q₂
WHERE    R.A=T.A       ⎦
```

$$\{r.A \mid r \in Q_1\} \cup \{r.A \mid r \in Q_2\}$$

Q_1 Q_2

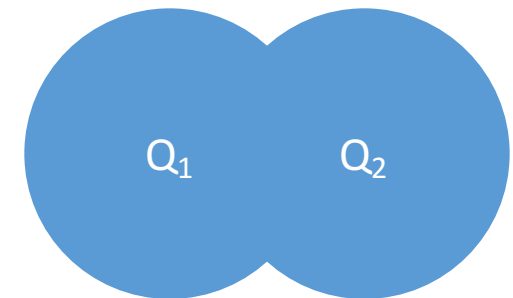# UNION

Why aren't there duplicates?

What if we want duplicates?

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

Q_1
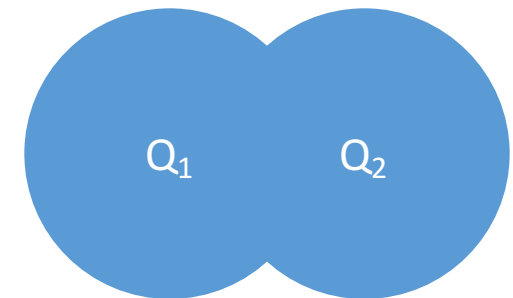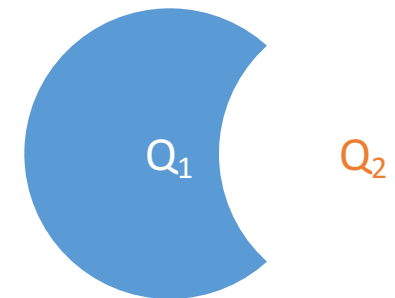
Q_2

$\{r.A | r \in Q_1\} \cup \{r.A | r \in Q_2\}$

Q_1 Q_2

UNION ALL

ALL variants of the other set operators as well

```
SELECT   R.A
FROM     R, S
WHERE    R.A=S.A
```
Q_1

```
UNION ALL
```

```
SELECT   R.A
FROM     R, T
WHERE    R.A=T.A
```
Q_2

$$\{r.A | r \in Q_1\} \cup \{r.A | r \in Q_2\}$$

Q_1 Q_2

EXCEPT

```
SELECT  R.A
FROM    R, S        ⎱ Q₁
WHERE   R.A=S.A
EXCEPT
SELECT  R.A
FROM    R, T        ⎱ Q₂
WHERE   R.A=T.A
```

$\{r.A | r \in Q_1\} / \{r.A | r \in Q_2\}$

Q_1 Q_2

63

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 gizmos in US **AND** China"*

What if two companies have HQ in US: BUT one has factory in China (but not US) and vice versa? **What goes wrong?**

64

Set Operators: Evaluation Order
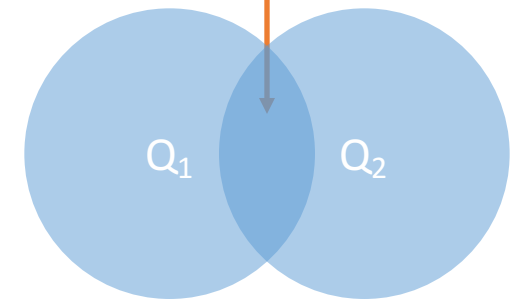
Note the evaluation order:

1. SFW queries (Q_1, Q_2)

2. **Then** the set op.
   1. *On the projected output sets!*

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

Q_1

Q_2

$\{r.A | r \in Q_1\} \cup \{r.A | r \in Q_2\}$

Q_1 Q_2

Must consider what tuples the set operation is **actually** executed on

65

One Solution: **Nested Queries**

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

```
SELECT  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')
```

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

66

Nested queries: Sub-queries Returning 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')
```

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

67

Nested Queries

Is this query equivalent?

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

Beware of duplicates!

68

Nested Queries

```
SELECT DISTINCT c.city
FROM    Company c,
        Product pr,
        Purchase p
WHERE   c.name = pr.maker
  AND   p.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

69

Subqueries Returning 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    Purchase
        WHERE   maker = 'Gizmo-Works')
```

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

70

Subqueries Returning Relations

You can also use operations of the form:

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

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)
```

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

71

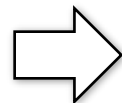# Nested queries as alternatives to INTERSECT and EXCEPT

Motivation: INTERSECT and EXCEPT not in some DBMSs!

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

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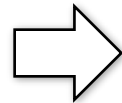
```
SELECT R.A, R.B
FROM   R
WHERE EXISTS(
        SELECT *
        FROM S
        WHERE R.A=S.A AND R.B=S.B)
```

If R, S have no duplicates, then can write without sub-queries (HOW?)

```
(SELECT R.A, R.B
 FROM   R)
EXCEPT
(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)
```

72

A question for Database Fans & Friends

- Can we express the previous nested queries as single SFW queries?

- Hint: show that all SFW queries are monotone (roughly: more tuples, more answers).
  - A query with **ALL** is not monotone

Correlated Queries

```
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!

Note also: this can still be expressed as single SFW 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)

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.

Activity-3-1.ipynb

2. Aggregation & GROUP BY

What you will learn about in this section

1. Aggregation operators

2. GROUP BY

3. GROUP BY: with HAVING, semantics

4. ACTIVITY: Fancy SQL Pt. I

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
```

Note: Same as COUNT(*).
Why?

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'
```
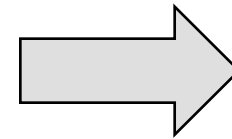
82

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 (= 20 + 30)

Grouping and Aggregation

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.
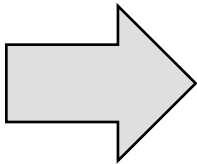
Let's see what this means...

84

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.
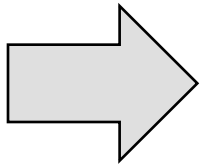
1. Compute the FROM and WHERE clauses

```
SELECT    product,
          SUM(price*quantity) AS TotalSales

FROM      Purchase
WHERE     date > '10/1/2005'
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

2. Group by the attributes in the GROUP BY
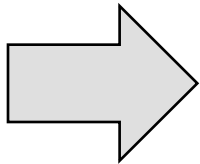
```
SELECT     product,
            SUM(price*quantity) AS TotalSales
FROM       Purchase
WHERE      date > '10/1/2005'
GROUP BY product
```

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

87

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	TotalSales
Bagel	50
Banana	15

GROUP BY v.s. Nested Quereis

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

```
SELECT DISTINCT x.product,
        (SELECT Sum(y.price*y.quantity)
          FROM   Purchase y
          WHERE  x.product = y.product
            AND y.date > '10/1/2005') AS TotalSales
FROM    Purchase x
WHERE   x.date > '10/1/2005'
```

HAVING Clause

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

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
```

Why?

- S = Can ONLY contain attributes $a_1,...,a_k$ and/or aggregates over other attributes

- C_1 = is any condition on the attributes in $R_1,...,R_n$

- C_2 = is any condition on the aggregate expressions

91

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. Evaluate FROM-WHERE: apply condition C_1 on the attributes in $R_1, ..., R_n$

2. GROUP BY the attributes $a_1, ..., a_k$

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

93

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

94

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 is <u>much</u> more efficient!

Activity-3-2.ipynb

3. Advanced SQL-izing

What you will learn about in this section

1. Quantifiers

2. NULLs

3. Outer Joins

4. ACTIVITY: Fancy SQL Pt. II

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 <u>some</u> products with price < 100

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

Existential: easy ! ☺

Quantifiers

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

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

A **<u>universal quantifier</u>** is a logical constant (roughly) of the form "for all"

Find all companies with products <u>all</u> having price < 100

⬇ Equivalent

Find all companies that make <u>only</u> products with price < 100

Universal: hard ! ☹

100

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**

  - 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 OR weight > 190)
```

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

Rule in SQL: include only tuples that yield TRUE

103

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!

RECAP: 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
```

Both equivalent:
Both INNER JOINS!

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

106

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 !

107

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

Note: another equivalent way to write an INNER JOIN!

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

110

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

Activity-3-3.ipynb

Summary

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