# Recap: Basic Structured Query Language (SQL)

SQL is a standard language for querying and manipulating data in RDBMS

- CREATE TABLE with PRIMARY KEYs/FOREIGN KEYs
- SFW query

# ICCS240 Database Management

# SQL
## (cont.)

Many slides in this lecture are either from or adapted from slides provided by

Theodoros Rekatsinas, UW-Madison

Abdu Alawini. UIUC

# Basic form of SQL Query

```
SELECT  <attributes: A1, ..., Am>
FROM    <one or more relations: R1, ..., Rk>
WHERE   <predicate/conditions: C>
```

Call this a select-from-where or SFW query.

The result of an SQL query is a relation.

You may run/try examples from sqlex_run1.sql along with the following slides

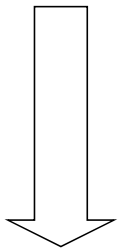
# Simple SQL Query: Selection

Selection ($\sigma$) 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'
```

$\sigma_{Category='Gadge\ '}(Product)$



| PName | Price | Category | Manufacturer |
|---|---|---|---|
| Gizmo | $19.99 | Gadgets | GizmoWorks |
| Powergizmo | $29.99 | Gadgets | GizmoWorks |

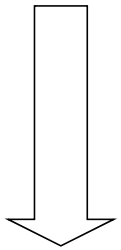
# Simple SQL Query: Projection

Projection (**Π**) 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'
```

$\Pi_{PName,Price,Manufacturer}(\sigma_{Category\ \ 'Gadgets'}(Product))$



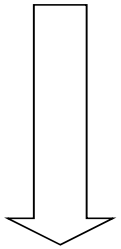
| PName | Price | Manufacturer |
|---|---|---|
| Gizmo | $19.99 | GizmoWorks |
| Powergizmo | $29.99 | GizmoWorks |

# SQL Query (may) create a result with a new schema

Input schema

Product(PName, Price, Category, Manfacturer)

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



Output schema

Answer(PName, Price, Manfacturer)

# A few tips about SQL

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

- **Values** are **not:**
  - Different: ‘Seattle’, ‘seattle’

- Use single quotes for constants:
  - ✓ ‘abc’
  - ✗ “abc”

# 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: eliminate duplicates

```
SELECT DISTINCT Category
FROM   Product
```

| Category |
| --- |
| Gadgets |
| Photography |
| Household |


```
SELECT Category
FROM   Product
```

| Category |
| --- |
| Gadgets |
| Gadgets |
| Photography |
| Household |

# ORDER BY: sorting the results

```
SELECT    PName, Price, Manufacturer
FROM      Product
WHERE     Category='gizmo' AND Price > 50
ORDER BY  Price, PName
```

Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.

# NULL VALUES

# VALUES CAN BE *NULL*

- Tuples in SQL relations can have *NULL* as a value for one or more components.

- *Comparison:*

When any value is compared with NULL, the truth value is *UNKNOWN*.

  - *Missing value*: e.g., we know 'Canon' company has some address, but we don't know what it is.
  - *Inapplicable*: e.g., the value of attribute *spouse* for an unmarried person.

# Comparison of NULLs and values

- The logic of conditions in SQL is really 3-valued logic:

  **TRUE, FALSE, UNKNOWN**

- *Comparison:*

  When any value is compared with NULL, the truth value is **UNKNOWN**

- *Outcome:*

  A query only produces a tuple in the answer

  if its truth value for the WHERE clause is TRUE (not FALSE or UNKNOWN)

# Three-Valued Logic

- Think of TRUE = 1, FALSE = 0, and UNKNOWN = ½
- AND = MIN(); OR = MAX(), NOT(*x*) = 1-*x*

- Example:

TRUE **AND** (FALSE **OR NOT(**UNKNOWN**)**)

=

Try **`SELECT TRUE AND (FALSE OR NOT(NULL));`**

# Tricky Example!

Sell

| bar | beer | price |
|---|---|---|
| Joe | Asahi | NULL |

```
SELECT  bar
FROM    Sell
WHERE   price < 2.00   OR   price >= 2.00
```


Surprising Result?

# 2-Valued Laws ≠ 3-Valued Laws

Some common laws, like the commutativity of AND, hold in 3-valued logic.

*p* AND *q* ≡ *q* AND *p*

But many others do not!

Example: (complement?)

In 2-valued law: *p* OR NOT *p* ≡ TRUE

In 3-valued law: When *p* = UNKNOWN,
L.H.S. is MAX( ½, (1 – ½ )) = ½ = UNKNOWN

# Leverage relevant issue by testing for NULL

Instead of relying on there not being NULLs or getting the default behavior for NULLs, can override this.

Test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

```
SELECT bar
FROM   Sell
WHERE  price < 2.00 OR price >= 2.00
           OR  price IS NULL
```

# Multi-Relation query

# Foreign Key Constraints

Suppose we have the following schema:

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

**Students**

| sid | name | gpa |
|---|---|---|
| 101 | Bob | 3.2 |
| 123 | Mary | 3.8 |

**Enrolled**

| student_id | cid | grade |
|---|---|---|
| 123 | 564 | A |
| 123 | 537 | A+ |

*Q: student_id alone is not a key- what is?*

And we want to impose the following constraint:

'Only bona fide students may enroll in courses' i.e. a student must appear in the Students table to enroll in a class

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(sid)
)
```

# Foreign keys & Update operations

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

What if we insert a tuple into Enrolled, but no corresponding student?

INSERT is rejected (foreign keys are constraints)!

What if we delete a student?

1. Disallow the delete (`ON DELETE NO ACTION`) DEFAULT!
2. Remove all of the courses for that student (`ON DELETE CASCADE`)
3. *SQL allows via NULL* (`ON DELETE SET NULL`)
4. SQL allows via default values (`ON DELETE SET DEFAULT`)

*(similar idea for update?)*

# JOINs of multiple relations

```
Product     (prname, price,  category, maker)
Company     (cname,  stock,  country)
Purchase    (buyer,  seller, store,    product)
Person      (pname,  phoneNumber, city)
```

Find all products under $100 manufactured in USA; return their name, price and its maker.

Several equivalent ways to write a basic join in SQL:

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

```
SELECT prname, price, maker
FROM   Product, Company
WHERE  maker = cname
       AND  country = 'USA'
       AND  price <= $100
```

```
SELECT prname, price, maker
FROM   Product JOIN Company
       ON maker = cname
WHERE  price <= $100  AND  country = 'USA'
```

# Example

Product2

| PName | Price | Category | Manuf |
|---|---|---|---|
| Gizmo | $119 | Gadgets | GWorks |
| PowerGizmo | $29 | Gadgets | GWorks |
| SingleTouch | $49 | Photography | Micron |
| MultiTouch | $203 | Household | Hitachi |

Company2

| Cname | Stock | Country |
|---|---|---|
| GWorks | 25 | USA |
| Micron | 65 | USA |
| Hitachi | 15 | Japan |

```
SELECT pname, price, manuf
FROM   Product2, Company2
WHERE  manuf = cname
       AND  country = 'USA'
       AND  price <= $100
```

| PName | Price | Maker |
|---|---|---|
| PowerGizmo | $29 | Gworld |
| SingleTouch | $49 | Micron |

# Ambiguity in Multi-Relation

```
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??

# Disambiguating in Multi-Relation

```
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
```

*Tip: Always prefix with relation name to make it clear/easier to read. But optional!*

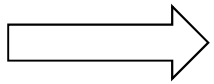
# Naming Convention

- To avoid ambiguities, every attribute name has two components

**Relation . Attribute**

- When there is no ambiguity, one can drop the relation name component

```
SELECT  person.name, person.income
FROM    person
WHERE   person.age < 30
```



```
SELECT  name, income
FROM    person
WHERE   age < 30
```

# Your playtime:

Download and run **sqlex_data2.sql**

From the dataset:

- From the table **person,** compute a new table by selecting only the persons with an income between 20 and 30, and adding an attribute that has, for every tuple, the same value as **income**.
- The fathers of persons who earn more than 20K
- Father and mother of every person
- Persons that earn more than their father, showing name, income, and income of the father

MotherChild

| mother | child |
|---|---|
| Lisa | Mary |
| Lisa | Greg |
| Anne | Kim |
| Anne | Phil |
| Mary | Andy |
| Mary | Rob |

FatherChild

| father | child |
|---|---|
| Steve | Frank |
| Greg | Kim |
| Greg | Phil |
| Frank | Andy |
| Frank | Rob |

Data from Werner Nutt

Person

| name | age | income |
|---|---|---|
| Andy | 27 | 21 |
| Rob | 25 | 15 |
| Mary | 55 | 42 |
| Anne | 50 | 35 |
| Phil | 26 | 30 |
| Greg | 50 | 40 |
| Frank | 60 | 20 |
| Kim | 30 | 41 |
| Mike | 85 | 35 |
| Lisa | 75 | 87 |

# Operational Semantics

(Similar to all single-relation queries)

1. Start with the product of all the relations in the FROM clause.
2. Apply the selection condition from the WHERE clause.
3. Project onto the list of attributes and expressions in the SELECT clause.

# Meaning (Semantics) of SQL Queries

```
SELECT   A1, A2, …, Ak
FROM     R1, R2, …, Rn
WHERE    conditions
```

Translate to Relational Algebra

$$\Pi_{A_1,A_2,\ldots,A_k}\left(\sigma_{\text{conditions}}(R_1 \times R_2 \times \cdots \times R_n)\right)$$

Select-From-Where queries are precisely Join-Select-Project

# Meaning (Semantics) of SQL Queries

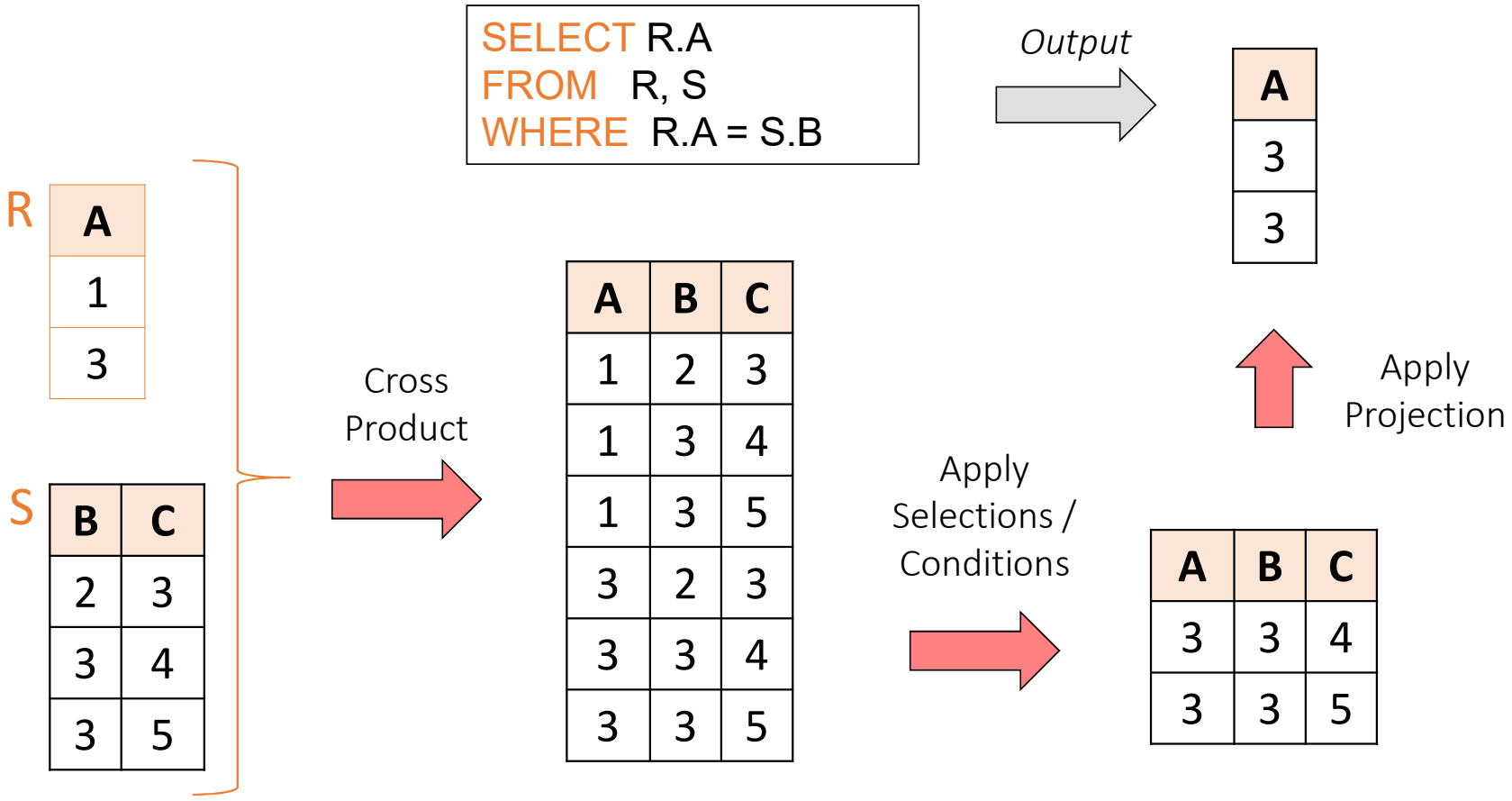
```
SELECT   A1, A2, …, Ak
FROM     R1, R2, …, Rn
WHERE    conditions
```

Nested Loop

```
Answer = {}
for x1 in R1 do
   for x2 in R2 do
      …
      for xn in Rn do
         if Conditions
            then Answer = Answer U {(A1,…,Ak)}
return Answer
```

Almost never the *fastest* way to compute it!

# An example of SQL semantics



# Operational Semantics: more explanation

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

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)}

1. (From) Take **cross product**:

$$X = R \times S$$

2. (Where) Apply **selections / conditions**: = Filtering!

$$Y = \{(r, s) \in X \mid r.A == r.B\}$$

3. (Select) Apply **projections** to get final output: = Returning only *some* attributes

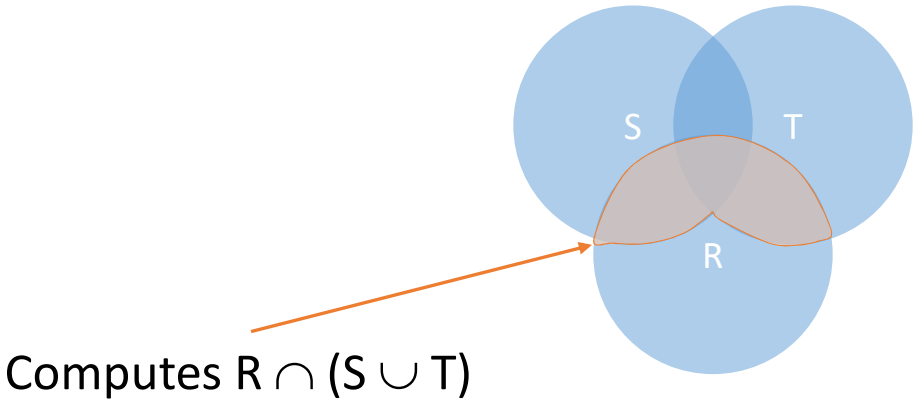
$$Z = (y.A, ) \; for \; y \in Y$$

Remembering this order is critical to understanding the output of certain 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?




But what if S = Ø ?

Go back to the semantics!