Data Relationships & Joins

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. Multiple Database Tables: Why Bother?

2. Using Multiple Tables To Model Relationships

3. Joins

Some material from lectures created by Professor Greiner (used with permission), some material from lectures created by Professor Kedem (used with permission)

Context

- We are now SQL experts ☺
- Well . . . we're getting close to "good, intermediate SQL programmers"
- Today: address the very important topic of JOIN
- Because the topic is so important, we'll spend some time to motivate why we need the JOIN operation in the first place
- After all: perfectly good NOSQL databases don't surface this capability to clients!

Multiple Database Tables: Why Bother?

- As its name connotes, the JOIN "glues" related, but separate tables together
- This raises the question: if they're related, why separate them in the first place ☺
- The relational database answer: by separating data "intelligently", we ensure that
  - No redundant data (the DRY principle)
  - Database normalization theory tells us precisely how reducing specific types of redundancy ensures that we eliminate entire classes of error scenarios
  - We will study normalization theory later in the course …
- For now: stick with the intuition that "DRY is good"
- Examples on next slides …

Single table has redundant data.
- Opportunity for inconsistency
- More storage space & I/O time

Product

prod_name	prod_price	prod_manufacturer	prod_address	prod_city	prod_state
Gizmo	$19.99	GizmoWorks	123 Gizmo St.	Houston	TX
Powergizmo	$39.99	GizmoWorks	123 Gizmo St.	Huston	TX
Widget	$19.99	WidgetsRUs	20 Main St.	New York	NY
HyperWidget	$203.99	Hyper	1 Mission Dr.	San Francisco	CA

Multiple tables can eliminate redundancy.

Product

prod_name	prod_price	prod_manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$39.99	GizmoWorks
Widget	$19.99	WidgetsRUs
HyperWidget	$203.99	Hyper

Manufacturer

man_name	man_addre	man_city	man_state
GizmoWorks	123 Gizmo St.	Houston	TX
WidgetsRUs	20 Main St.	New York	NY
Hyper	1 Mission Dr.	San Francisco	CA

“Intelligent” Separation

- Q: OK, I get that separating *product* from *manufacturer* data is a good idea …
  - But: doesn’t that separation make it more likely that we’ll make mistakes?
  - Now we can make mistakes in two places rather than just one ☹
    - Meaning: since the tables are related, an error in one table’s data implicitly causes an error in the other table!
- A: we’re not going to completely separate the tables
  - We’ll maintain “links” between the tables to represent the fact that the data are related
  - And enforce the semantics of those relationships

- Q: what do we call these “links”?
- A: foreign keys ☺

Foreign keys & referential integrity

Product

prod_name	prod_price	prod_manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$39.99	GizmoWorks
Widget	$19.99	WidgetsRUs
HyperWidget	$203.99	Hyper

PK FK

Manufacturer

man_name	man_address	man_city	man_state
GizmoWorks	123 Gizmo St.	Houston	TX
WidgetsRUs	20 Main St.	New York	NY
Hyper	1 Mission Dr.	San Francisco	CA

PK

Product's manufacturer is a *foreign key*.

- FK refers to & depends on a PK. Must have same number and type of columns.
- We want to enforce *referential integrity* – each prod_manufacturer is a man_name.

Creating a table with a foreign key

```
CREATE TABLE Manufacturer (
   man_name TEXT,
   man_address TEXT,
   man_city TEXT,
   man_state CHAR(2),
   PRIMARY KEY (man_name)
);
```

Product's FK depends on Manufacturer's PK.
Product's definition depends on Manufacturer's definition.

```
CREATE TABLE Product (
   prod_name TEXT,
   prod_price MONEY,
   prod_manufacturer TEXT,
   PRIMARY KEY (prod_name),
   FOREIGN KEY (prod_manufacturer) REFERENCES Manufacturer (man_name)
);
```

Enforcing referential integrity

Product

prod_name	prod_price	prod_manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$39.99	GizmoWorks
Widget	$19.99	WidgetsRUs
HyperWidget	$203.99	Hyper
Thingy	$105.99	ThingMaker

Manufacturer

man_name	man_address	man_city	man_state
GizmoWorks	123 Gizmo St.	Houston	TX
WidgetsRUs	20 Main St.	New York	NY
Hyper	1 Mission Dr.	San Francisco	CA

Foreign keys vs. pointers

Product

prod_name	prod_price	prod_manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$39.99	GizmoWorks
Widget	$19.99	WidgetsRUs
HyperWidget	$203.99	Hyper

Manufacturer

man_name	man_address	man_city	man_state
GizmoWorks	123 Gizmo St.	Houston	TX
WidgetsRUs	20 Main St.	New York	NY
Hyper	1 Mission Dr.	San Francisco	CA

FK repeats data. Pointer points at a location.

- Pointers are difficult to maintain when data changes.
- Pointers are uni-directional.

Using Multiple Tables To Model Relationships

Introduction

- Very important to understand that when we refer to "relationships" ...
  - There are different types of relationships
  - The different types reflect fundamental modeling assumptions about the data
- As I've said previously: the data in front of you is just a "point in time" statement
- The data don't capture your modeling assumptions about the data characteristics or the data model
- Obvious examples:
  - The range of values that an attribute can have
  - The "fact" that a *product* must be associated with a *manufacturer*
- We refer to these assumptions as meta-data, and one of the strengths of relational databases is they allow us to specify and enforce such meta-data

Binary Relations

- In mathematics, a binary relation over sets X and Y is a subset of the Cartesian product $X \times Y$
  - In other words: a binary relation is a set of ordered pairs $(x, y) : x \in X, y \in Y$
- Sometimes we can say something useful about the "functionality" of $X \Leftrightarrow Y$
- Example:
  - X is the set of all people
  - Y is the set of all countries
  - R is the BornIn relationship: "Person x was born in country y"
- Your data modeler informs you that *"a person can be born in at most one country"*
- Q: what are the implications of that meta-data?

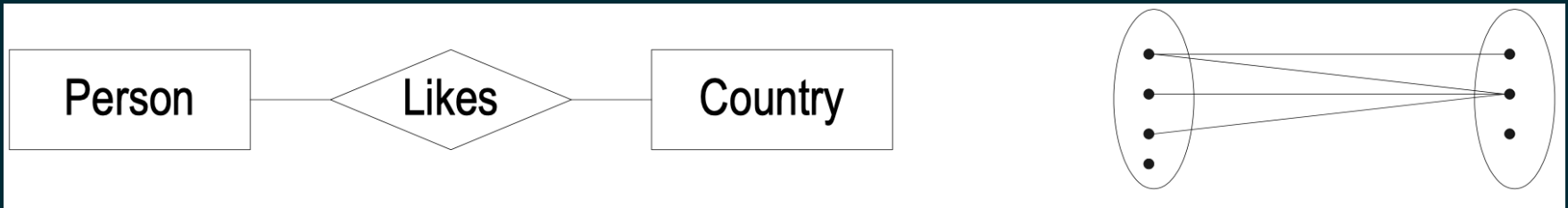
“Many-to-One” Relationships

- The BornIn relationship is many-to-one from *X* to *Y iff* for each element of *X* (people) there exists at most one element of *Y* (countries) related to it
  - Our data modeler has told us that this relationship is “many-to-one”
- For BornIn relationship we’re saying:
  - Every person was born in at most one country
  - Important: but it’s possible for a person not to be born in any country!
    - Example: birth occurs on a ship in international waters, or in the International Space Station
  - So: the relationship is really a partial function

“One-to-One” Relationships

- Now consider the HeadsCountry relationship between X (people) and Y (country)
  - A person can be the “head of state” for at most one country
  - A country can have at most one “head of state”
- We say that the HeadOf relationship is one-to-one between X and Y *iff* for each element of X (people) there exists at most one element of Y (countries) related to it and for each element of Y there exists at most one element of X related to it
- Therefore: a relationship R is “one-to-one” *iff*
  - R is “many-to-one” from $X \to Y$
  - R is “many-to-one” from $Y \to X$

“Many-to-Many” Relationships



- Now consider the PersonLikes relationship between X (people) and Y (countries)
- The PersonLikes relationship is called “many-to-many” between X and Y *iff*
  - R is not “many-to-one” between $X \to Y$
  - R is not “many-to-one” between $Y \to X$
- We’re saying that the PersonLikes relationship is not a function!
  - A person may “like” multiple countries
  - A country may be “liked” by multiple people
  - Note: this is not the “partial function” issue in which some person simply doesn’t like any country

Relationship Cardinality Is Not A Property of the "Data"

- Repeating one more time … ☺
- Saying that, for example, that data have a "one-to-one" relationship, is not a statement about specific data
- Instead: it's a statement about all possible instances of the data
  - Alternatively: it's a statement about immutable characteristics of the domains under discussion
- Example: a list of "authors and books" may happen to describe books with only one author
  - Inspecting the data will show that every *book tuple* refers to only one *author* tuple …
  - Nevertheless: the relationship is "many-to-many"!
    - A given book can certainly be written by multiple authors
    - A given author can certainly write multiple books

Segue To Database Tables

- As we're about to see, we can structure our database tables to reflect & enforce the relationship cardinalities that we've just discussed
- Let's resume the "products & manufacturers" example ...

> *"Item Description" Pattern*
> *The item description pattern consists of an "item" object (i.e., an object of the class "item") and an "item description" object. An "item description" object has attribute values which may apply to more than one "item" object; an "item" object has its own individual assignment of attribute values.*
>
> *Peter Coad, CACM 1992*

One-to-many / many-to-one

Manufacturer

man_name	man_address	man_city	man_state
GizmoWorks	123 Gizmo St.	Houston	TX
WidgetsRUs	20 Main St.	New York	NY
Hyper	1 Mission Dr.	San Francisco	CA
Gadgets	10 Minimax St.	Houston	TX

Product

prod_name	prod_price	prod_manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$39.99	GizmoWorks
Widget	$19.99	WidgetsRUs
HyperWidget	$203.99	Hyper

1-to-many: Each Manufacturer can have many Products.

many-to-1: Each Product is made by exactly one Manufacturer.

many-to-1 = OOP's Item-Description Pattern

Lookup tables – common use of one-to-many

Lookup table

Ref. integrity ensures man_state is in this list.

State

state
AL
…
TX
WV

Manufacturer

man_name	man_address	man_city	man_state
GizmoWorks	123 Gizmo St.	Houston	TX
WidgetsRUs	20 Main St.	New York	NY
Hyper	1 Mission Dr.	San Francisco	CA
Gadgets	10 Minimax St.	Houston	TX
Wowza	99 Whynot Rd.	Palo Alto	CA

1-to-many: Each State can have many Manufacturers.

many-to-1: Each Manufacturer is in one State.

Many-to-many

Student

st_id	first_name	last_name
S01	John	Smith
S02	Mary	Wallace
S03	Sue	Roper
S04	Mark	Jones

Enrollment

st_id	crn	grade
S01	0123	A
S01	4611	C
S02	0123	B

Junction table

Course

crn	dept	number
0123	COMP	140
1513	COMP	160
4611	ELEC	220

Each Student can be enrolled in many Courses.

Each Course has many Students.

```
CREATE TABLE Enrollment (
   …
   PRIMARY KEY (st_id, crn),
   FOREIGN KEY (st_id) REFERENCES Student (st_id),
   FOREIGN KEY (crn) REFERENCES Course (crn)
);
```

Many-to-many – alternative

Student

st_id	first_name	last_name
S01	John	Smith
S02	Mary	Wallace
S03	Sue	Roper
S04	Mark	Jones

Enrollment

en_id	st_id	crn	grade
1	S01	0123	A
2	S01	4611	C
3	S02	0123	F
4	S02	0123	B

Junction table

Course

crn	dept	number
0123	COMP	140
1513	COMP	160
4611	ELEC	220

Each Student can be enrolled in many Courses.

Each Course has many Students.

```
CREATE TABLE Enrollment (
   …
   PRIMARY KEY (en_id),
   FOREIGN KEY (st_id) REFERENCES Student (st_id),
   FOREIGN KEY (crn) REFERENCES Course (crn)
);
```

Segue To Joins

- I hope that you're now persuaded that "refactoring data" into separate tables is a good idea
- More than a matter of the DRY principle!
- The approach allows us to model the fundamental relationship(s) between the two domains
- Q: but now that we've "refactored" the tables, how do "glue them back together"
  - Example: we need information about a *product* and its *manufacturer*
- A: the magic of the JOIN operator

Joins

Quick Refresher

- JOIN operations take two relations and return another relation
- A JOIN operation is a Cartesian product enhanced to require that tuples in the two relations "match" under some condition
- We've been using the JOIN operator for a long time!
  - It's the "comma operator" in the FROM clause below ☺

```
1  SELECT name, course_id
2    FROM students, takes
3    WHERE student.ID = takes.ID;
```

- There are various flavors of JOIN "types" and various flavors of JOIN "conditions"

Natural Join

Compare this classic SELECT-FROM-WHERE query:

- "Raw" Cartesian product version

```
SELECT name, course_id
    FROM instructor, teaches
    WHERE instructor.ID = teaches.ID
```

- To the natural join version

```
SELECT name, course_id
    FROM instructor NATURAL JOIN teaches
```

- With a natural join, tuples are joined only if they have the same values for all shared attributes
  - Returns a relation containing only one copy of each common attribute
- Any number of relations can be joined in a single expression

Danger! (I)

- *"List the names of students along with the titles of courses that they have taken"*

```
1  SELECT name, title
2    FROM student NATURAL JOIN takes NATURAL JOIN course;
```

- Q: is this implementation correct?
- A: unfortunately not ☹
  - *student*:

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32

  - *takes*:

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-

  - STUDENT NATURAL JOIN TAKES:

ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C

  - so far, so good ...

Danger! (II)

```
1  SELECT name, title
2    FROM student NATURAL JOIN takes NATURAL JOIN course;
```

- *"List the names of students along with the titles of courses that they have taken"*
- STUDENT NATURAL JOIN TAKES:

ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C

- But *course* contains these attributes: *course_id, title, dept_name, credits*
- A natural join between the *student* $\times$ *takes* relation and the *course* relation requires that the tuples match on both *dept_name* and *course_id*
  - In other words: this implementation will not include tuples in which a student takes a course in a department other than the student's own department
  - Given our department's "non-cs" requirements, this will make many students very unhappy ☹

Danger! (III)

```
1  SELECT name, title
2    FROM student NATURAL JOIN takes, course
3    WHERE takes.course_id = course.course_id;
```

- *"List the names of students along with the titles of courses that they have taken"*
- The above implementation is correct
- The USING syntax is one way to prevent the nasty bug we've just discussed
  - While still using the JOIN syntax ...
  - The USING clause allows you to explicitly specify the attributes to be "equated" during the JOIN operation

```
1  SELECT name, title
2     FROM (instructor NATURAL JOIN takes)
3     JOIN course using (course_id)
```

Using on To “Hard-Code” Join Condition

These are all equivalent

```
1       -- Output relation will have 2 ID attributes
2       SELECT *
3          FROM student JOIN takes ON student.ID = takes.ID
```

```
1       -- These relations share ID attribute
2       -- Will only emit one ID attribute
3       SELECT *
4       FROM student NATURAL JOIN takes
```

```
1       -- Example shows that ‘‘on’’ can be replaced
2       -- with ‘‘where’’ clause
3       SELECT * from student, takes
4       WHERE student.ID = takes.ID
```

- Can use the on condition to specify a general (arbitrary) predicate on the relations
- on behaves like where condition for this join version
- Will behave differently for outer join (stay tuned)

My Recommendation

- Avoid NATURAL JOIN: you're asking for trouble ☺
- POSTGRESQL: use INNER JOIN along with explicit ON specification
  - Inner join requires tuples in the joined relation to have the same value on the join predicate
  - Example below

```
1 SELECT suppliers.supplier_id, suppliers.supplier_name,
      orders.order_date
2 FROM suppliers
3 INNER JOIN orders
4 ON suppliers.supplier_id = orders.supplier_id
```

- Note: NATURAL JOIN is not a join "type" (such as INNER or OUTER joins)
- Rather: NATURAL JOIN is a join condition (such as ON or USING)
  - In other words: "syntactic sugar" ☺

Motivating "Outer" Joins

Find all students, and include their "course taken" information

- Will this implementation do the job?

```
SELECT * FROM student NATURAL JOIN takes
```

- Problem: students are allowed to "have not taken courses yet"
  - Example: student "Snow" in *students* in "small university" example, does not appear in *takes*
- Natural or inner join will not include such students!
- In general: use outer joins when natural joins would lose tuples that can't be matched across relations
  - Outer joins will include these tuples, showing the lack of "linked information" *via* NULL values
  - One approach: add "Snow tuple" to output relation, take all attributes from *student*, set all attributes from *takes* to NULL

Small-Scale Outer Join Example

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

Figure: *course* Relation

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Figure: *prereq* Relation

Observe:

- course relation is missing information for CS-347
- prereq relation is missing information about CS-315
- Queries using “natural join” will be unable to match on these tuples

Left Outer Join

```
course natural left outer join prereq
```

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

- left outer join only preserves tuples in the relation named before the left outer join operation
  - Preserves tuples to the left of the outer join operation
  - Inserts NULL in each column from the right side of the join operation if it can't find a "matching" tuple in the right-hand-side relation

Right Outer Join

```
1    course natural right outer join prereq
```

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

- right outer join only preserves tuples in the relation named after the right outer join operation
  - Preserves tuples to the right of the outer join operation
  - Inserts NULL in each column from the left side of the join operation if it can't find a "matching" tuple in the left-hand-side relation

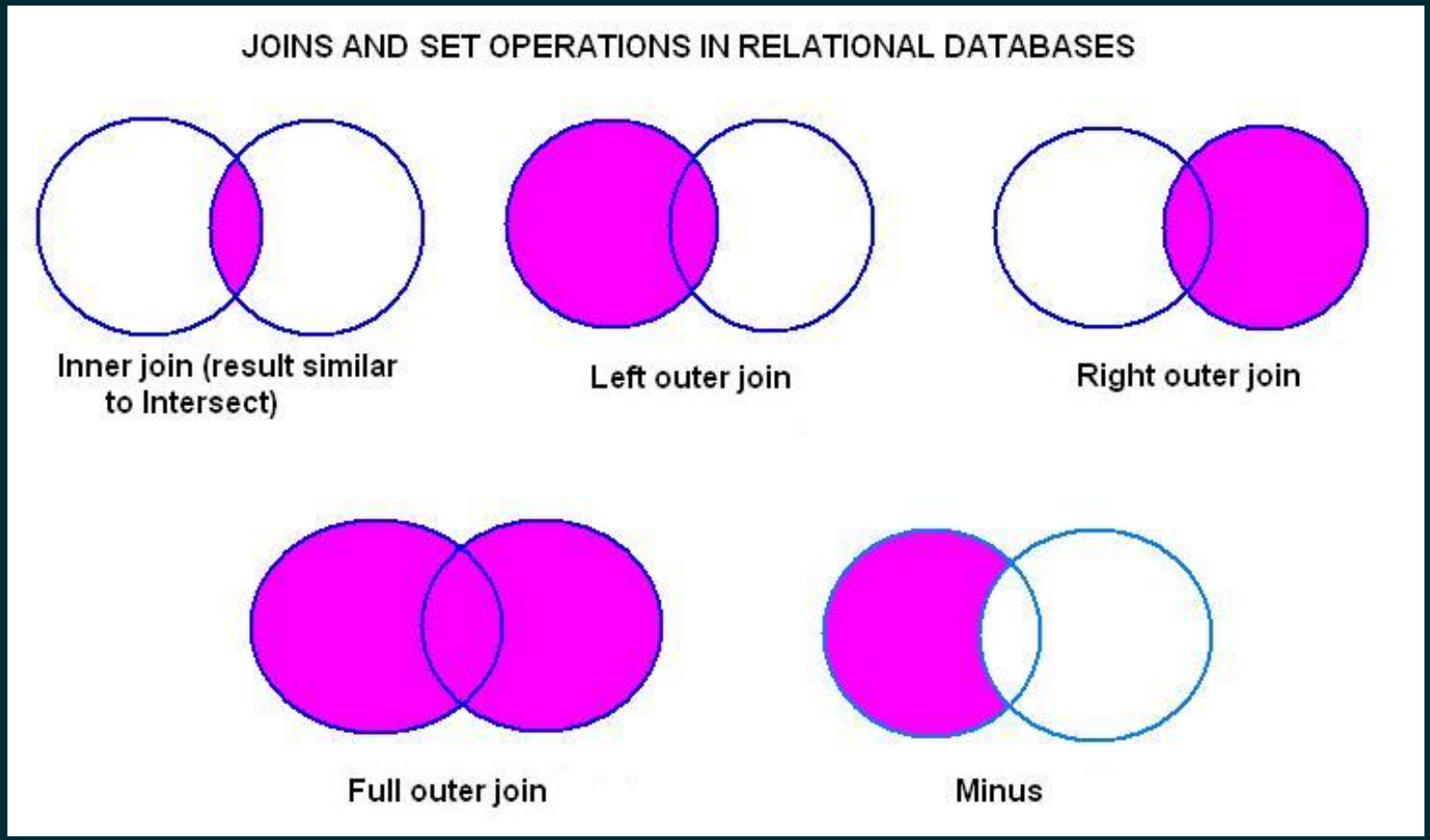
Full Outer Join

```
course natural full outer join prereq
```

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

- full outer join preserves tuples in both relations

Different Join Types As Venn Diagrams



- INNER JOIN: Select only those rows that have values in common in the columns specified in the ON clause.
- LEFT, RIGHT, FULL OUTER JOIN: Select all rows from the table on the left (or right, or both) regardless of whether the other table has values in common and (usually) enter NULL where data is missing

The on Clause & Inner Joins

Can use the on clause for inner joins as well as natural joins

```
1       course INNER JOIN prereq
2          ON course.course_id = prereq.course_id
```

course_id	*title*	*dept_name*	*credits*	*prere_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

Q: What is the difference between this “inner join” version, and a natural join?
A: the two versions are almost identical, except that “natural join” would eliminate the second occurrence of *course_id*

The on Clause & Outer Joins

Can use the on clause for outer joins as well as natural joins

```
course LEFT OUTER JOIN prereq
   ON course.course_id = prereq.course_id
```

course_id	*title*	*dept_name*	*credits*	*prere_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	*null*	*null*

Note: semantics for "use *versus* non-use" of ON is identical to the previous slide's "inner join" example

Joining tables – inner joins

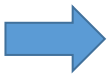
Product

prod_name	prod_price	prod_manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$39.99	GizmoWorks
Widget	$19.99	WidgetsRUs
HyperWidget	$203.99	Hyper

Manufacturer

man_name	man_address	man_city	man_state
GizmoWorks	123 Gizmo St.	Houston	TX
WidgetsRUs	20 Main St.	New York	NY
Hyper	1 Mission Dr.	San Francisco	CA

By far, the most common kind of join. See other kinds later.



prod_name	prod_price	prod_manufacturer	man_name	man_address	man_city	man_state
Gizmo	$19.99	GizmoWorks	GizmoWorks	123 Gizmo St.	Houston	TX
Powergizmo	$39.99	GizmoWorks	GizmoWorks	123 Gizmo St.	Houston	TX
Widget	$19.99	WidgetsRUs	WidgetsRUs	20 Main St.	New York	NY
HyperWidget	$203.99	Hyper	Hyper	1 Mission Dr.	San Francisco	CA

Joining tables – INNER JOIN

Product

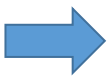
prod_name	prod_price	prod_manufacturer

Manufacturer

man_name	man_address	man_city	man_state

```
SELECT *
FROM Product
INNER JOIN Manufacturer ON prod_manufacturer = man_name;
```

```
SELECT *
FROM Company
INNER JOIN Product ON prod_manufacturer = man_name;
```



prod_name	prod_price	prod_manufacturer	man_name	man_address	man_city	man_state
Gizmo	$19.99	GizmoWorks	GizmoWorks	123 Gizmo St.	Houston	TX
Powergizmo	$39.99	GizmoWorks	GizmoWorks	123 Gizmo St.	Houston	TX
Widget	$19.99	WidgetsRUs	WidgetsRUs	20 Main St.	New York	NY
HyperWidget	$203.99	Hyper	Hyper	1 Mission Dr.	San Francisco	CA

Joining tables – INNER JOIN

Product

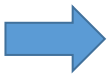
prod_name	prod_price	prod_manufacturer

Manufacturer

man_name	man_address	man_city	man_state

~~SELECT *~~
~~FROM Product, Manufacturer~~
~~WHERE prod_manufacturer = man_name;~~

Old style – deprecated & error-prone



prod_name	prod_price	prod_manufacturer	man_name	man_address	man_city	man_state
Gizmo	$19.99	GizmoWorks	GizmoWorks	123 Gizmo St.	Houston	TX
Powergizmo	$39.99	GizmoWorks	GizmoWorks	123 Gizmo St.	Houston	TX
Widget	$19.99	WidgetsRUs	WidgetsRUs	20 Main St.	New York	NY
HyperWidget	$203.99	Hyper	Hyper	1 Mission Dr.	San Francisco	CA

Resolving attribute name conflicts

Person (name, address, works_for)
Company (name, address)

```
SELECT Person.name, Person.address
FROM Person
INNER JOIN Company ON Person.works_for = Company.name;
```

```
SELECT p.name, p.address
FROM Person p
INNER JOIN Company c ON p.works_for = c.name;
```

Order of inner joins is irrelevant

```
SELECT *
FROM A
INNER JOIN B ON A.a = B.a
INNER JOIN C ON B.b = C.b
INNER JOIN D ON C.c = D.c
INNER JOIN E ON D.d = E.d;
```

Any organization of inner joins is semantically equivalent.
System will optimize speed.
You should maximize readability.

Order of operations – 1) Join

```
SELECT prod_name
FROM Product
INNER JOIN Manufacturer ON prod_manufacturer = man_name
WHERE state = 'TX';
```

Join condition

Filter condition

prod_name	prod_price	prod_manufacturer	man_name	man_address	man_city	man_state
Gizmo	$19.99	GizmoWorks	GizmoWorks	123 Gizmo St.	Houston	TX
Powergizmo	$39.99	GizmoWorks	GizmoWorks	123 Gizmo St.	Houston	TX
Widget	$19.99	WidgetsRUs	WidgetsRUs	20 Main St.	New York	NY
HyperWidget	$203.99	Hyper	Hyper	1 Mission Dr.	San Francisco	CA

Order of operations – 2) Filter

```
SELECT prod_name
FROM Product
INNER JOIN Manufacturer ON prod_manufacturer = man_name
WHERE state = 'TX';
```

prod_name	prod_price	prod_manufacturer	man_name	man_address	man_city	man_state
Gizmo	$19.99	GizmoWorks	GizmoWorks	123 Gizmo St.	Houston	TX
Powergizmo	$39.99	GizmoWorks	GizmoWorks	123 Gizmo St.	Houston	TX

Order of operations – 3) Project

```
SELECT prod_name
FROM Product
INNER JOIN Manufacturer ON prod_manufacturer = man_name
WHERE state = 'TX';
```

prod_name
Gizmo
Powergizmo

Multiple kinds of joins

Inner, outer, and cross joins

What about *dangling tuples*?

Manufacturer

man_name	man_state
GizmoWorks	TX
WidgetsRUs	NY
Hyper	CA
NewCo	TX

Product

prod_name	prod_man
Gizmo	GizmoWorks
Powergizmo	GizmoWorks
Widget	WidgetsRUs
HyperWidget	Hyper

man_name	man_state	prod_name	prod_man
GizmoWorks	TX	Gizmo	GizmoWorks
GizmoWorks	TX	Powergizmo	GizmoWorks
WidgetsRUs	NY	Widget	WidgetsRUs
Hyper	CA	HyperWidget	Hyper

Inner join omits NewCo, since it has no products.

```
SELECT  *
FROM Manufacturer
INNER JOIN Product ON man_name = prod_man;
```

Left outer join

Manufacturer

man_name	man_state
GizmoWorks	TX
WidgetsRUs	NY
Hyper	CA
NewCo	TX

Product

prod_name	prod_man
Gizmo	GizmoWorks
Powergizmo	GizmoWorks
Widget	WidgetsRUs
HyperWidget	Hyper

man_name	man_state	prod_name	prod_man
GizmoWorks	TX	Gizmo	GizmoWorks
GizmoWorks	TX	Powergizmo	GizmoWorks
WidgetsRUs	NY	Widget	WidgetsRUs
Hyper	CA	HyperWidget	Hyper
NewCo	TX	NULL	NULL

Includes dangling tuples from LEFT table.

```
SELECT  *
FROM Manufacturer
LEFT OUTER JOIN Product ON man_name = prod_man;
```

Right outer join

Manufacturer

man_name	man_state
GizmoWorks	TX
WidgetsRUs	NY
Hyper	CA
NewCo	TX

Product

prod_name	prod_man
Gizmo	GizmoWorks
Powergizmo	GizmoWorks
Widget	WidgetsRUs
HyperWidget	Hyper

man_name	man_state	prod_name	prod_man
GizmoWorks	TX	Gizmo	GizmoWorks
GizmoWorks	TX	Powergizmo	GizmoWorks
WidgetsRUs	NY	Widget	WidgetsRUs
Hyper	CA	HyperWidget	Hyper

Useless in this example.
Referential integrity guarantees
Product has no dangling tuples.

```
SELECT  *
FROM Manufacturer
RIGHT OUTER JOIN Product ON man_name = prod_man;
```

Right outer join

Manufacturer

man_name	man_state
GizmoWorks	TX
WidgetsRUs	NY
Hyper	CA
NewCo	TX

Product

prod_name	prod_man
Gizmo	GizmoWorks
Powergizmo	GizmoWorks
Widget	WidgetsRUs
HyperWidget	Hyper
NewThing	NULL

Assume no FK / referential integrity in this example.

man_name	man_state	prod_name	prod_man
GizmoWorks	TX	Gizmo	GizmoWorks
GizmoWorks	TX	Powergizmo	GizmoWorks
WidgetsRUs	NY	Widget	WidgetsRUs
Hyper	CA	HyperWidget	Hyper
NULL	NULL	NewThing	NULL

```
SELECT  *
FROM Manufacturer
RIGHT OUTER JOIN Product ON man_name = prod_man;
```

Outer joins are symmetric

Left outer join A,B = Right outer join B,A

Stylistically, LEFT OUTER JOIN is much more commonly used.

Full outer join

Manufacturer

man_name	man_state
GizmoWorks	TX
WidgetsRUs	NY
Hyper	CA
NewCo	TX

Product

prod_name	prod_man
Gizmo	GizmoWorks
Powergizmo	GizmoWorks
Widget	WidgetsRUs
HyperWidget	Hyper
NewThing	NULL

Again, no FK.

man_name	man_state	prod_name	prod_man
GizmoWorks	TX	Gizmo	GizmoWorks
GizmoWorks	TX	Powergizmo	GizmoWorks
WidgetsRUs	NY	Widget	WidgetsRUs
Hyper	CA	HyperWidget	Hyper
NewCo	TX	NULL	NULL
NULL	NULL	NewThing	NULL

Include dangling tuples from both tables.

```
SELECT  *
FROM Manufacturer
FULL OUTER JOIN Product ON man_name = prod_man;
```

Cross join

Manufacturer

man_name	man_state
GizmoWorks	TX
WidgetsRUs	NY
Hyper	CA

Product

prod_name	prod_man
Gizmo	GizmoWorks
Powergizmo	GizmoWorks
Widget	WidgetsRUs
HyperWidget	Hyper

```
SELECT  *
FROM Manufacturer
CROSS JOIN Product;
```

No join condition.

Old, deprecated style:

~~SELECT *~~
~~FROM Product, Manufacturer;~~

All combinations of records!
Cross-product of tables.

man_name	man_state	prod_name	prod_man
GizmoWorks	TX	Gizmo	GizmoWorks
WidgetsRUs	NY	Gizmo	GizmoWorks
Hyper	CA	Gizmo	GizmoWorks
GizmoWorks	TX	Powergizmo	GizmoWorks
WidgetsRUs	NY	Powergizmo	GizmoWorks
Hyper	CA	Powergizmo	GizmoWorks
GizmoWorks	TX	Widget	WidgetsRUs
WidgetsRUs	NY	Widget	WidgetsRUs
Hyper	CA	Widget	WidgetsRUs
GizmoWorks	TX	HyperWidget	Hyper
WidgetsRUs	NY	HyperWidget	Hyper
Hyper	CA	HyperWidget	Hyper

Some special cases

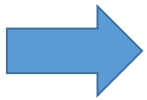
Of inner & outer joins

Examples adapted from Coding Horror.

Self-joins – Joining table with itself

Employee

id	name	boss_id
1	Joe	3
2	Mary	3
3	Charles	4
4	Lisa	NULL



name	boss_name
Joe	Charles
Mary	Charles
Charles	Lisa

```
SELECT emp.name, boss.name AS boss_name
FROM Employee emp
INNER JOIN Employee boss ON emp.boss_id = boss.id;
```

Useful when one column (boss_id) refers to another column's (id) data.

Equi-joins & non-equi-joins

Employee

id	name	wage
1	Joe	$10
2	Mary	$11
3	Charles	$10
4	Lisa	NULL

e1.id	e2.id
1	1
1	3
2	2
3	1
3	3

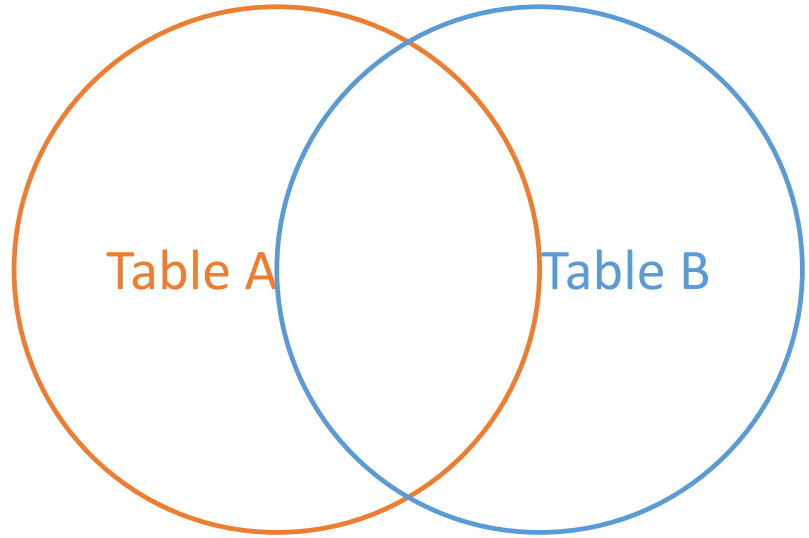
```
SELECT e1.id, e2.id
FROM Employee e1
INNER JOIN Employee e2
   ON e1.wage = e2.wage;
```

e1.id	e2.id
2	1
2	3

```
SELECT e1.id, e2.id
FROM Employee e1
INNER JOIN Employee e2
   ON e1.wage > e2.wage;
```

Visual summary

Of inner & outer joins



Examples adapted from Coding Horror.

A

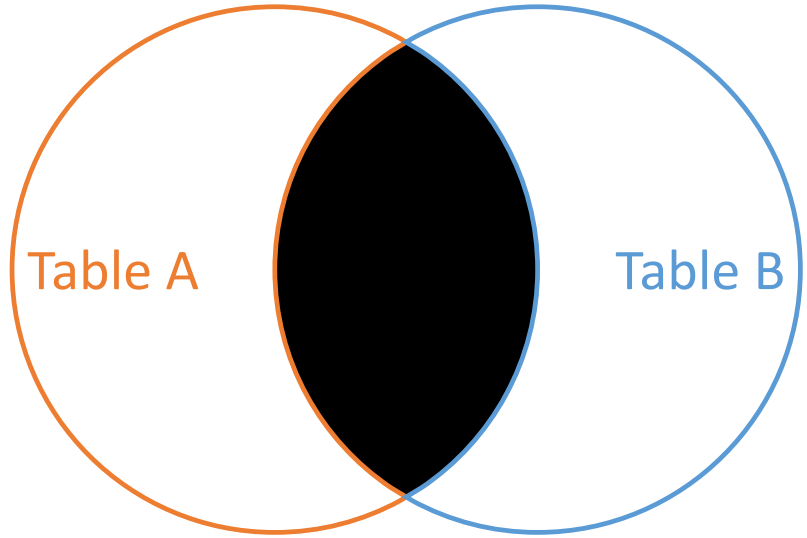
id	data
1	A
2	B
3	C
4	D

B

id	data
2	W
4	X
6	Y
8	Z

```
SELECT *
FROM A
INNER JOIN B ON A.id = B.id;
```

id	A.data	B.data
2	B	W
4	D	X



Records matching both A and B.

A

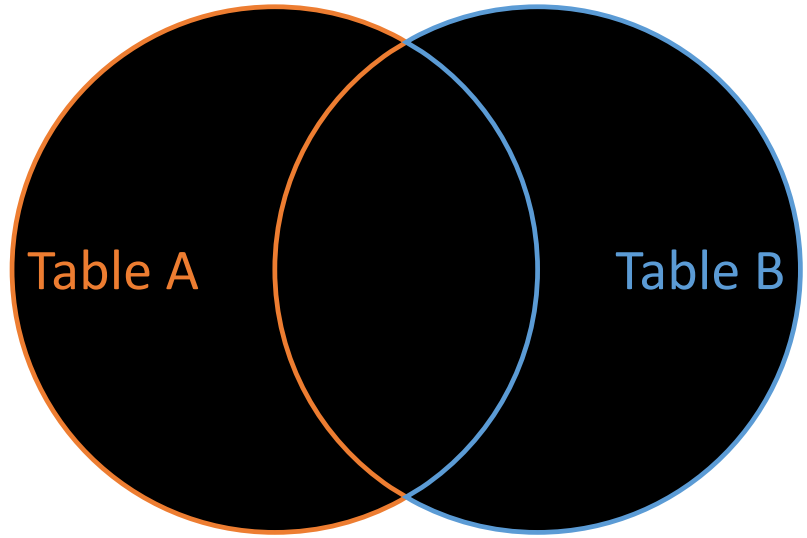
id	data
1	A
2	B
3	C
4	D

B

id	data
2	W
4	X
6	Y
8	Z

id	A.data	B.data
1	A	
2	B	W
3	C	
4	D	X
6		Y
8		Z

```
SELECT *
FROM A
FULL OUTER JOIN B ON A.id = B.id;
```



Records matching A or B.

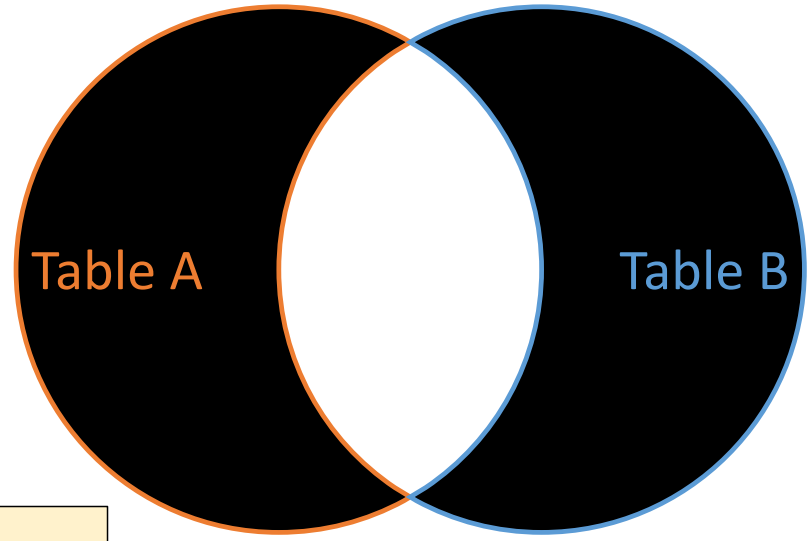
A

id	data
1	A
2	B
3	C
4	D

B

id	data
2	W
4	X
6	Y
8	Z

id	A.data	B.data
1	A	
3	C	
6		Y
8		Z




```
SELECT *
FROM A
FULL OUTER JOIN B ON A.id = B.id
WHERE A.data IS NULL OR B.data IS NULL;
```

Records matching either A or B, but not both.

A

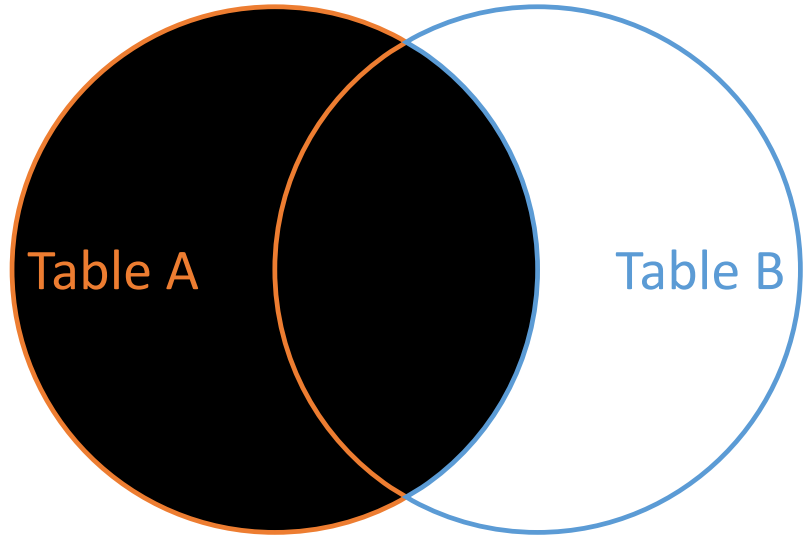
id	data
1	A
2	B
3	C
4	D

B

id	data
2	W
4	X
6	Y
8	Z

id	A.data	B.data
1	A	
2	B	W
3	C	
4	D	X

```
SELECT *
FROM A
LEFT OUTER JOIN B ON A.id = B.id;
```



Records matching A.

A

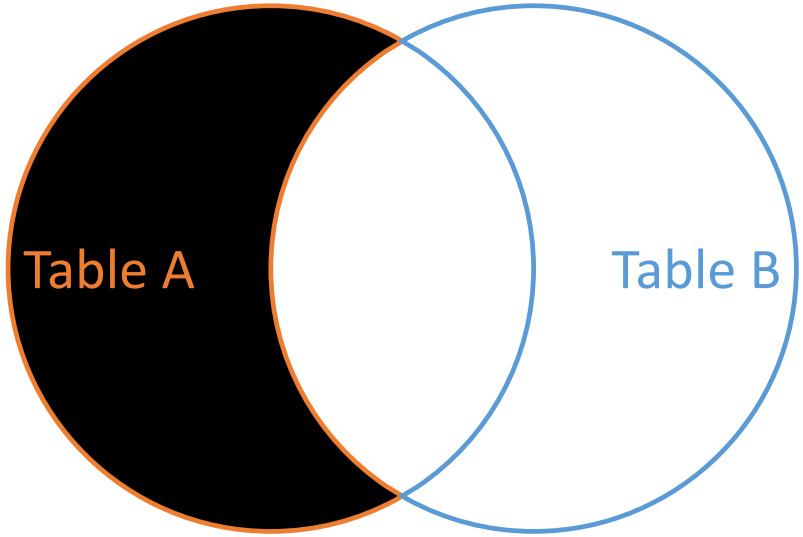
id	data
1	A
2	B
3	C
4	D

B

id	data
2	W
4	X
6	Y
8	Z

id	A.data	B.data
1	A	
3	C	

```
SELECT *
FROM A
LEFT OUTER JOIN B ON A.id = B.id
WHERE B.data IS NULL;
```



Records matching only A.

Today's Lecture: Wrapping it Up

Multiple Database Tables: Why Bother?

Using Multiple Tables To Model Relationships

Joins

Readings

- Textbook discusses *Join Expressions* in Chapter 4