W4111 – Introduction to Databases
Section 002, Fall 2021

Lecture 4: ER, Relational, SQL (III)

COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

Contents

Contents

- Questions, answers, discussion.

- A worked example involving:
  - ER modeling.
  - SQL DDL and data model creation.
  - Top-down and bottom-up modeling.

- A little more relational algebra and the dreaded “RelaX Calculator.”

- SQL Insert, Update and Delete.

Questions, Answers, Discussion

Systematic Treatment of NULL

From Ed Discussion:

> **"Confusion:** When looking over the W4111_hw1_material.ipynb file, it's mentioned that "isDead is either true or NULL. NULL means unknown or not applicable". But, in the spec for the HW, it says that the column should strictly take 'Y' or 'N' values. The example output also generated confusion since it did include 'N' in the isDead column.
>
> **Question:** Regardless, I understand any column can take NULL values but I just wanted to make sure we should only include 'Y' and NULL as values in our column where the latter is instead of 'N'. Should we use 'true' rather than 'Y' as the HW 1 materials file specifies as well?"

Comments:

- IMHO the question is focusing on your understanding type constraints and data cleanup. So, "Y" or "N" is fine with me.
- That aside, my solution would be *true* or NULL.
  - The data does indicate is someone is dead, but ... ...
  - The data is "old." I cannot be sure someone is alive. They could have just died.

Codd's 12 Rules

Rule 1: Information Rule

The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.

Rule 2: Guaranteed Access Rule

Every single data element (value) is guaranteed to be accessible logically with a combination of table-name, primary-key (row value), and attribute-name (column value). No other means, such as pointers, can be used to access data.

Rule 3: Systematic Treatment of NULL Values

The NULL values in a database must be given a systematic and uniform treatment. This is a very important rule because a NULL can be interpreted as one the following – data is missing, data is not known, or data is not applicable.

Rule 4: Active Online Catalog

The structure description of the entire database must be stored in an online catalog, known as data dictionary, which can be accessed by authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

Rule 5: Comprehensive Data Sub-Language Rule

A database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation.

Rule 6: View Updating Rule

All the views of a database, which can theoretically be updated, must also be updatable by the system.

Codd's 12 Rules

Rule 7: High-Level Insert, Update, and Delete Rule

A database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to yield sets of data records.

Rule 8: Physical Data Independence

The data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications.

Rule 9: Logical Data Independence

The logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. For example, if two tables are merged or one is split into two different tables, there should be no impact or change on the user application. This is one of the most difficult rule to apply.

Rule 10: Integrity Independence

A database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface.

Rule 11: Distribution Independence

The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems.

Rule 12: Non-Subversion Rule

If a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.

Codd's 12 Rules

Rule 3: Systematic treatment of null values:

- "Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing **missing information** and **inapplicable** information in a systematic way, independent of data type."

- Sometimes programmers and database designers are tempted to use "special values" to indicate unknow, missing or inapplicable values.
  - String: "", "NA", "UNKNOWN", …
  - Numbers: -1, 0, -9999

- Indicators can cause confusion because you have to carefully code some SQL statements to the specific, varying choices programmers made.

NULL and Correct Answers

```
In [4]: 1 %sql describe aaaaS21Examples.null_examples;
```

```
 * mysql+pymysql://dbuser:***@localhost
3 rows affected.
```

Out[4]:

Field	Type	Null	Key	Default	Extra
name	varchar(32)	NO	PRI	None	
weight	int	YES		None	
net_worth	int	YES		None	

```
In [5]: 1 %sql select * from aaaaS21Examples.null_examples;
```

```
 * mysql+pymysql://dbuser:***@localhost
4 rows affected.
```

Out[5]:

name	weight	net_worth
Joe	100	100
Larry	0	0
Pete	None	None
Tim	200	200

Without NULL, to get a correct answer:

- I must understand the domain to determine “unknown” values or know what choice a developer made.
- Explicitly include “where weight != 0” in all statements.
- And this varies from column to column, table to table, schema to schema, etc.

```
In [7]: 1 %%sql select avg(weight) as avg_weight, avg(net_worth) as avg_net_worth
        2     from aaaaS21Examples.null_examples where name in ('Joe', 'Larry', 'Tim')
```

```
 * mysql+pymysql://dbuser:***@localhost
1 rows affected.
```

Out[7]:

avg_weight	avg_net_worth
100.0000	100.0000

```
In [9]: 1 %%sql select avg(weight) as avg_weight, avg(net_worth) as avg_net_worth
        2     from aaaaS21Examples.null_examples where name in ('Joe', 'Pete', 'Tim')
```

```
 * mysql+pymysql://dbuser:***@localhost
1 rows affected.
```

Out[9]:

avg_weight	avg_net_worth
150.0000	150.0000

Back to Our Excellent Question

- "When *deathYear* is null do we assume that the person is alive? What if he is dead and information is just missing?"
- Well,
  - My interpretation is that if there is no death date, *isDead* should be NULL.
  - I might expect that the person is dead if their current age would be 125, but that is just a guess.
- Also,
  - Having a column *isDead* is actually a bad design pattern in this case.
  - The value of *isDead* is *functionally dependent* on *deathYear.*
  - Database designs avoid *functional dependencies* to prevent *update anomalies.* Some could forget to update both columns when learning a fact.
  - In other scenarios, I might know that a person *isDead* but not know the *deathDate.* So, this design decision is problem/scenario specific.

Other Questions?

Worked Example:

1. *ER Model from scratch.*
2. *Schema definition, DDL, ... and some new concepts.*
3. *Where's the data?*

Worked Example – Top Down

- Scenario
  - Course with complex course ID.
  - Section
  - Faculty
  - Department
  - Instructor – Department Assoc. entity with properties (role, date).
  - Instructor – Section
  - Student - Section

- Do ER (live) diagram in Lucidchart.

- Do schema creation
  - In DataGrip
  - Show copying statements int the notebook.

```
CREATE TABLE `cu_model`.`courses` (
  `dept_code` VARCHAR(4) NOT NULL,
  `faculty_code` ENUM('BC', 'C', 'W', 'E', 'G') NOT NULL,
  `credit_level` ENUM('0', '1', '2', '3', '4', '6', '8', '9') NOT NULL,
  `course_no` VARCHAR(3) NOT NULL,
  `title` VARCHAR(64) NOT NULL,
  `full_course_no` VARCHAR(12) GENERATED ALWAYS AS
      (concat(dept_code,faculty_code,credit_level,course_no)),
  PRIMARY KEY (`dept_code`, `faculty_code`, `credit_level`, `course_no`));
```

Bottom-Up: Some Sources of Information

- Course Codes:
  https://www.cc-seas.columbia.edu/sites/dsa/files/handbooks/Columbia%20Key%20to%20Course%20Listing.pdf
- Course/Section Properties:
  http://www.columbia.edu/cu/bulletin/uwb/
- Common Search Criteria → Indexes, Query Parameters, Data Links
  https://doc.search.columbia.edu/classes/Ferguson?instr=&name=&days=&semes=&hour=&moi=
- Department Information:
  - Script: https://www.columbia.edu/content/academics/departments
  - Codes: https://academic-admin.cuit.columbia.edu/dept_code
- Specific Information:
  - Course and Instructor: https://opendataservice.columbia.edu/api/9/json
  - Vergil Data: https://vergil.registrar.columbia.edu/feeds/cw.js
  - Vergil Search: https://vergil.registrar.columbia.edu/doc-adv-queries.php?key=ferguson&moreresults=2

Meet in the Middle

- Show cu_info project with data and processing.
- Show notebook for loading the data.
- Show getting part of the way through parsing HTML in project

Some More Relational Algebra

Union Operation

- The union operation allows us to combine two relations
- Notation: $r \cup s$
- For $r \cup s$ to be valid.
  1. *r, s* must have the *same* **arity** (same number of attributes)
  2. The attribute domains must be **compatible** (example: 2nd column of *r* deals with the same type of values as does the 2nd column of *s*)
- Example: to find all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or in both

$$\Pi_{course\_id}(\sigma_{semester=\text{“Fall”} \wedge year=2017}(section)) \cup \Pi_{course\_id}(\sigma_{semester=\text{“Spring”} \wedge year=2018}(section))$$

Union Operation (Cont.)

- Result of:

$$\Pi_{course\_id}(\sigma_{semester=\text{“Fall”} \wedge year=2017}(section)) \cup \Pi_{course\_id}(\sigma_{semester=\text{“Spring”} \wedge year=2018}(section))$$

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Note: The preloaded dataset on the RelaX calculator is different from the most recent data referenced in the book. It is from a previous edition.

Set-Intersection Operation

- The set-intersection operation allows us to find tuples that are in both the input relations.
- Notation: $r \cap s$
- Assume:
  - r, s have the *same arity*
  - attributes of r and s are compatible
- Example: Find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters.

$$\Pi_{course\_id}(\sigma_{semester=\text{“Fall”} \wedge year=2017}(section)) \cap \Pi_{course\_id}(\sigma_{semester=\text{“Spring”} \wedge year=2018}(section))$$

  - Result

course_id
CS-101

Set Difference Operation

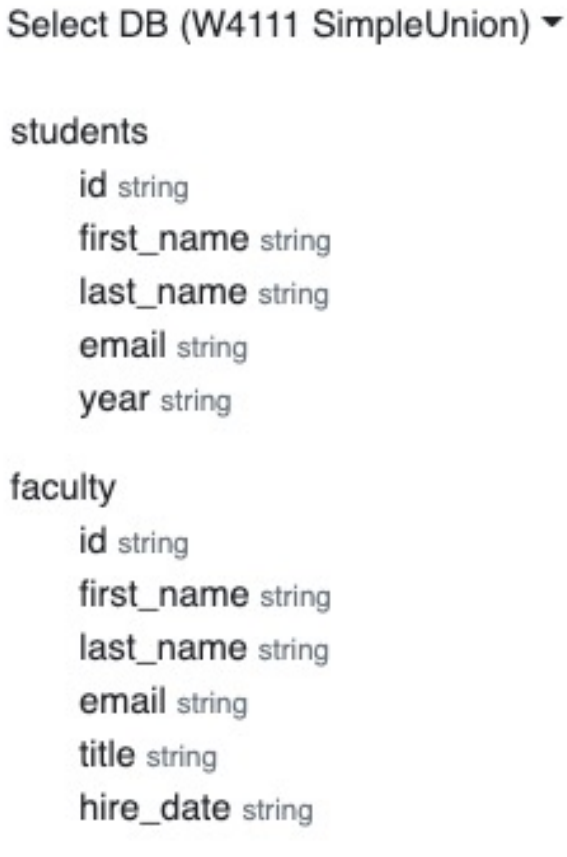
- The set-difference operation allows us to find tuples that are in one relation but are not in another.
- Notation $r - s$
- Set differences must be taken between **compatible** relations.
  - r and s must have the same arity
  - attribute domains of r and s must be compatible
- Example: to find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester

$$\Pi_{course\_id}(\sigma_{semester="Fall" \wedge year=2017}(section)) - \Pi_{course\_id}(\sigma_{semester="Spring" \wedge year=2018}(section))$$

course_id
CS-347
PHY-101

Same “arity”

- Same “arity”
  - Same number of columns.
  - Compatible types.
  - The i-th column in each table is from a compatible domain.
    - Student 5th column is “year.”
    - Faculty 5th column is “title”
    - Both are strings but combing them does not make sense.
- You can shape two incompatible tables using *project operations.* For example
  - π first_name,last_name,email (students)
    ∩
    π first_name,last_name,email (faculty)
  - π last_name,email, title←'Student' (students)
    ∪
    π last_name,email, title (faculty)



The Assignment Operation

- It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables.
- The assignment operation is denoted by ← and works like assignment in a programming language.
- Example: Find all instructor in the "Physics" and Music department.

$$Physics \leftarrow \sigma_{dept\_name="Physics"}(instructor)$$

$$Music \leftarrow \sigma_{dept\_name="Music"}(instructor)$$

$$Physics \cup Music$$

- With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.

The Rename Operation

- The results of relational-algebra expressions do not have a name that we can use to refer to them. The rename operator, ρ, is provided for that purpose
- The expression:

  $$\rho_x(E)$$

  returns the result of expression E under the name x
- Another form of the rename operation:

  $$\rho_{x(A1,A2,\ldots An)}(E)$$

Note: Assignment and rename can act a little wonky when using the calculator.

Equivalent Queries

- There is more than one way to write a query in relational algebra.
- Example: Find information about courses taught by instructors in the Physics department with salary greater than 90,000
- Query 1

$$\sigma_{dept\_name=\text{“Physics”} \wedge salary > 90{,}000}(instructor)$$

- Query 2

$$\sigma_{dept\_name=\text{“Physics”}}(\sigma_{salary > 90.000}(instructor))$$

- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.

Equivalent Queries

- There is more than one way to write a query in relational algebra.
- Example: Find information about courses taught by instructors in the Physics department
- Query 1

$$\sigma_{dept\_name=\text{“Physics”}}(instructor \bowtie_{instructor.ID = teaches.ID} teaches)$$

- Query 2

$$(\sigma_{dept\_name=\text{“Physics”}}(instructor)) \bowtie_{instructor.ID = teaches.ID} teaches$$

- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.

Insert, Update, Delete

Modification of the Database

- Deletion of tuples from a given relation.
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation

Deletion

- Delete all instructors

  delete from *instructor*

- Delete all instructors from the Finance department

  delete from *instructor*
  where *dept_name*= 'Finance';

- *Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building.*

  delete from *instructor*
  where *dept name* **in** (**select** *dept name*
  from *department*
  where *building* = 'Watson');

Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

```
delete from instructor
where salary < (select avg (salary)
                from instructor);
```

  - Problem: as we delete tuples from *instructor*, the average salary changes
  - Solution used in SQL:
    1. First, compute **avg** (salary) and find all tuples to delete
    2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Insertion

- Add a new tuple to *course*

  insert into *course*
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- or equivalently

  insert into *course* (*course_id*, *title*, *dept_name*, *credits*)
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- Add a new tuple to *student* with *tot_creds* set to null

  insert into *student*
  values ('3003', 'Green', 'Finance', *null*);

Insertion (Cont.)

- Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000.

```
insert into instructor
    select ID, name, dept_name, 18000
    from   student
    where  dept_name = 'Music' and total_cred > 144;
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.

Otherwise queries like

```
insert into table1 select * from table1
```

would cause problem

Updates

- Give a 5% salary raise to all instructors

```
update instructor
  set salary = salary * 1.05
```

- Give a 5% salary raise to those instructors who earn less than 70000

```
update instructor
  set salary = salary * 1.05
 where salary < 70000;
```

- Give a 5% salary raise to instructors whose salary is less than average

```
update instructor
set salary = salary * 1.05
where salary < (select avg (salary)
                from instructor);
```

Updates (Cont.)

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  - Write two **update** statements:

```
update instructor
   set salary = salary * 1.03
   where salary > 100000;
update instructor
    set salary = salary * 1.05
    where salary <= 100000;
```

  - The order is important
  - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before but with case statement

```
update instructor
set salary = case
                when salary <= 100000 then salary * 1.05
                else salary * 1.03
             end
```

Updates with Scalar Subqueries

- Recompute and update tot_creds value for all students

```
update student S
set tot_cred = (select sum(credits)
                from takes, course
                where takes.course_id = course.course_id  and
                      S.ID= takes.ID.and
                            takes.grade <> 'F' and
                       takes.grade is not null);
```

- Sets *tot_creds* to null for students who have not taken any course
- Instead of **sum**(*credits*), use:

```
case
    when sum(credits) is not null then sum(credits)
    else 0
 end
```

Summary

- INSERT, UPDATE and DELETE are pretty straightforward.
- UPDATE and DELETE are very similar to SELECT
  - WHERE clause specifies which rows are affected.
  - The SELECT choose the columns to return.
  - The SET clause chooses and changes columns.
  - DELETE just removes the specified rows.
- INSERT, UPDATE and DELETE changes must not violate constraints, e.g.
  - INSERT a row that causes a duplicate key.
  - DELETE a referenced (target) foreign key.
  - UPDATE columns that create a duplicate key.
  - INSERT values do not include all NOT NULL columns.
- I am not going to do example now, but you have seen and will see me do examples in the context of larger examples.

Insert, Update, Delete

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the **from** clause
- Three types of joins:
  - Natural join
  - Inner join
  - Outer join

Notes:

- You will also hear terms like equi-join, non-equi-join, theta join, semi-join, … …
- I ask for definitions on exams, but you can just look them up.

Natural Join in SQL

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column.
- List the names of instructors along with the course ID of the courses that they taught
  - **select** *name*, *course_id*
    from *students, takes*
    where *student.ID = takes.ID*;
- Same query in SQL with “natural join” construct
  - **select** *name*, *course_id*
    from *student* **natural join** *takes*;

Natural Join in SQL (Cont.)

- The **from** clause can have multiple relations combined using natural join:

  select $A_1, A_2, \ldots A_n$
  from r_1 **natural join** r_2 **natural join .. natural join** r_n
  where P ;

Student Relation

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Takes Relation

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	*null*

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
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2017	A
19991	Brandt	History	80	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	CS-101	1	Fall	2017	F
45678	Levy	Physics	46	CS-101	1	Spring	2018	B+
45678	Levy	Physics	46	CS-319	1	Spring	2018	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2017	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2018	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2017	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2018	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2017	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2018	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2017	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2018	*null*

Dangerous in Natural Join

- Beware of unrelated attributes with same name which get equated incorrectly
- Example -- List the names of students' instructors along with the titles of courses that they have taken
  - Correct version

```
select name, title
 from student natural join takes, course
 where takes.course_id = course.course_id;
```

  - Incorrect version

```
select name, title
 from student natural join takes natural join course;
```

    - This query omits all (student name, course title) pairs where the student takes a course in a department other than the student's own department.
    - The correct version (above), correctly outputs such pairs.

Natural Join with Using Clause

- To avoid the danger of equating attributes erroneously, we can use the "**using**" construct that allows us to specify exactly which columns should be equated.
- Query example

  select *name*, *title*
  from (*student* **natural join** *takes*) **join** *course* **using** (*course_id*)

Join Condition

- The **on** condition allows a general predicate over the relations being joined
- This predicate is written like a **where** clause predicate except for the use of the keyword **on**
- Query example

  select *
  from *student* **join** *takes* **on** *student_ID = takes_ID*

  – The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.

- Equivalent to:

  select *
  from *student , takes*
  where *student_ID = takes_ID*

Join Condition (Cont.)

- The **on** condition allows a general predicate over the relations being joined.
- This predicate is written like a **where** clause predicate except for the use of the keyword **on**.
- Query example

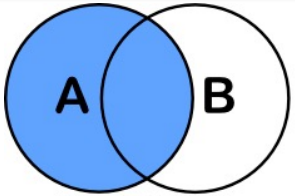
  select *
  from *student* **join** *takes* **on** *student_ID = takes_ID*

  – The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.

- Equivalent to:

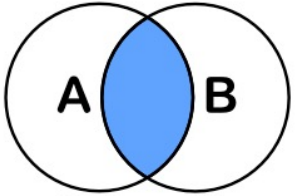
  select *
  from *student , takes*
  where *student_ID = takes_ID*

One Way to Think About Joins

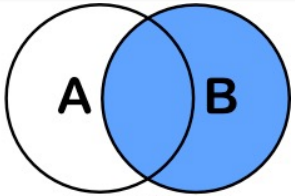


SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key

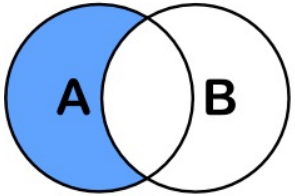
CHEATSHEET
SQL
JOINS



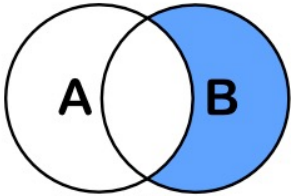
SELECT <auswahl>
FROM tabelleA A
INNER JOIN tabelleB B
ON A.key = B.key



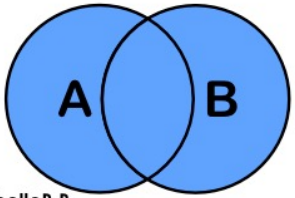
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key



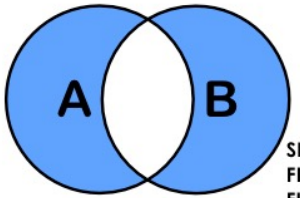
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
WHERE B.key IS NULL



SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL



SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key



SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL

Let's Do Some Example

- Theta Join: Find people in Lahman's DB who appeared (played) *after* Managing.
  - Show in Jupyter Notebook

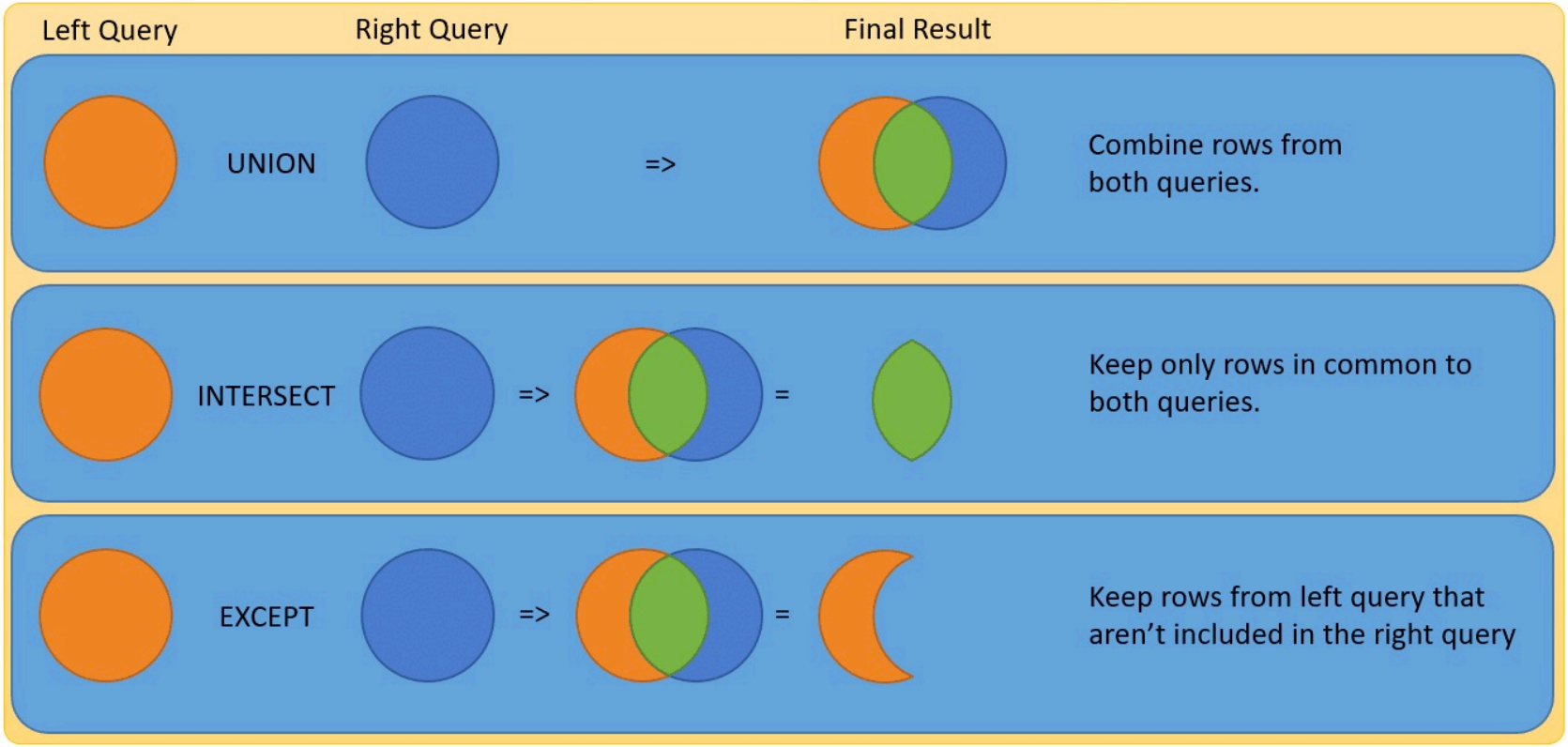
- Players who played for the Boston Redsox in 1967, and their Hall of Fame information if it exists.
  - Show in Jupyter Notebook

- Emulating full outer join:
  - The basic id is union left join and a right join
  - (select * from x left join y) union (select * from x right join y)

Set Operations

Set Operations

Visual Explanation of UNION, INTERSECT, and EXCEPT operators



Set Operations (Cont.)

- Set operations **union**, **intersect**, and **except**
  - Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the
  - **union all**,
  - **intersect all**
  - **except all**.

NOTE:

- SELECT implementing a project can have duplicate rows. If you do not want duplicates, you must use the DISTINCT key word.
- UNION behaves the other way. It removes duplicates. If you want to keep the duplicates, you have to select UNION ALL.
- Some SQL engines do not implement INTERSECT and/or EXCEPT, you can implement the function with subqueries, which we will cover soon.

Set Operations

- Find courses that ran in Fall 2017 or in Spring 2018

```
(select course_id  from section where sem = 'Fall' and year = 2017)
union
(select course_id  from section where sem = 'Spring' and year = 2018)
```

- Find courses that ran in Fall 2017 and in Spring 2018

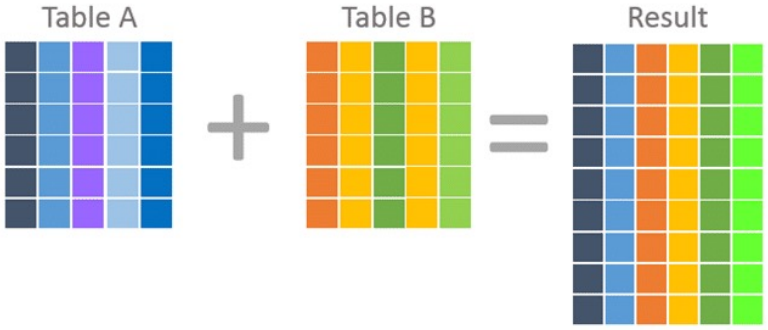
```
(select course_id  from section where sem = 'Fall' and year = 2017)
intersect
(select course_id  from section where sem = 'Spring' and year = 2018)
```

- Find courses that ran in Fall 2017 but not in Spring 2018

```
(select course_id  from section where sem = 'Fall' and year = 2017)
except
(select course_id  from section where sem = 'Spring' and year = 2018)
```

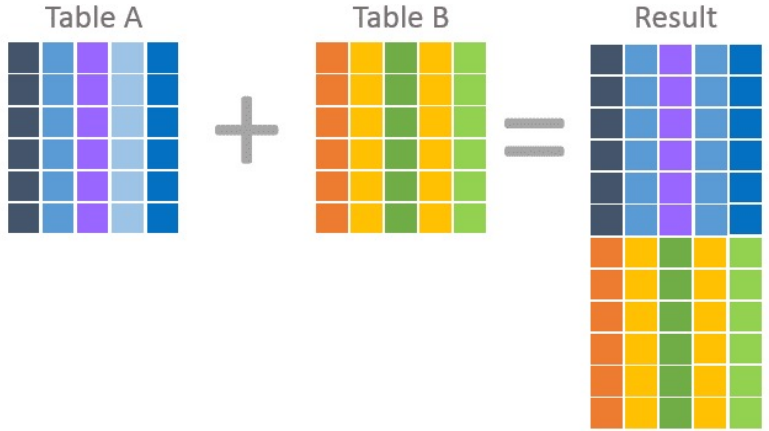
JOIN and UNION – A Final Word

Here is a visual depiction of a join. Table A and B's columns are combined into a single result.



Joins Combine Columns

Now compare the above depiction with that of a union. In a union, each row within the result is from one table OR the other. In a union, columns aren't combined to create results, rows are combined.



Unions Combine Rows

- UNION vs JOIN can be confusing. Basically,
  - JOIN puts the tables together "side by side."
  - Union puts the tables together "one on top of the oter."

Sub-Query

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **select-from-where** expression that is nested within another query.
- The nesting can be done in the following SQL query

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  as follows:

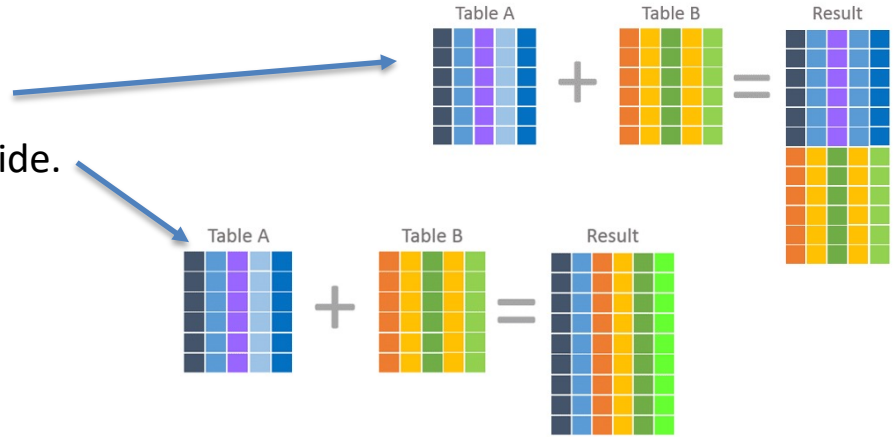
  - **From clause:** r_i can be replaced by any valid subquery
  - **Where clause:** P can be replaced with an

Note:

- This is a little cryptic.
- I think I know what they mean.
- There are some operations we will see later in the material, e.g IN, EXISTS, … …

Nested Subquery

- The slides that come with the book have surprisingly little material on nested subqueries.
- The concept is:
  - Extremely important.
  - Students often find subqueries more confusing than joins.
  - The relationship/difference of subqueries to joins is often, initial unclear.
- We have seen:
  - Union sort of puts a table on top of a table.
  - Join puts tables sort of puts tables side-by-side.
  - Subquery enables one query to call another during execution like a subfunction.



Consider Some Tables

Takes

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	None

Student

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Consider a Subquery Tables

select *, (select name from student where student.id=takes.id) as name from takes;

- Assume I wrote a function find_student_name(x)
  - Input is an x
  - Loops through all students and returns students with student.ID = x.
- The query with a subquery above is like:

```
result = []
For t in takes:
        new_r = t + find_student_name(t.id)
        result.append(new_r)
```

Switch to Notebook

Takes

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	None

Student

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **select-from-where** expression that is nested within another query.
- The nesting can be done in the following SQL query

  select $A_1, A_2, \ldots, A_n$
  from $r_1, r_2, \ldots, r_m$
  where P

  as follows:

  - **From clause:** r_i can be replaced by any valid subquery
  - **Where clause:** P can be replaced with an

Note: Subquery MUST return
- A single scalar if in the SELECT.
- A Table if in the FROM.
- If in the WHERE:
  - Either a scalar or a table.
  - Depending on the operation.

Set Membership

- Find courses offered in Fall 2017 and in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
        course_id in (select course_id
                      from section
                      where semester = 'Spring' and year= 2018);
```

- Find courses offered in Fall 2017 but not in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
        course_id not in (select course_id
                          from section
                          where semester = 'Spring' and year= 2018);
```

Set Membership (Cont.)

- Name all instructors whose name is neither "Mozart" nor Einstein"

```
select distinct name
from instructor
where  name not in ('Mozart', 'Einstein')
```

- Find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 10101

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in
(select course_id, sec_id, semester, year
from teaches
where teaches.ID= 10101);
```

Set Comparison – “some” Clause

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

- Same query using > some clause

```
select name
from instructor
where salary > some (select salary
                     from instructor
                     where dept name = 'Biology');
```

Definition of "some" Clause

- F <comp> **some** $r \Leftrightarrow \exists\ t \in r$ such that (F <comp> t)

  Where <comp> can be: $<,\ \leq,\ >,\ =,\ \neq$

  (5 < **some** | 0 / 5 / 6 |) = true (read: 5 < some tuple in the relation)

  (5 < **some** | 0 / 5 |) = false

  (5 = **some** | 0 / 5 |) = true

  (5 ≠ **some** | 0 / 5 |) = true (since 0 ≠ 5)

  (= **some**) ≡ **in**

  However, (≠ **some**) ≢ **not in**

Set Comparison – “all” Clause

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

```
select name
from instructor
where salary > all (select salary
                    from instructor
                    where dept name = 'Biology');
```

Definition of "all" Clause

- F <comp> **all** $r \Leftrightarrow \forall\ t \in r$ (F <comp> *t)*

(5 < **all** | 0 / 5 / 6 |) = false

(5 < **all** | 6 / 10 |) = true

(5 = **all** | 4 / 5 |) = false

(5 ≠ **all** | 4 / 6 |) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**
However, (= **all**) ≠ **in**

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

Use of “exists” Clause

- Yet another way of specifying the query “Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester”

```
select course_id
from section as S
where semester = 'Fall' and year = 2017 and
        exists  (select *
                 from section as T
                 where semester = 'Spring' and year= 2018
                                 and S.course_id = T.course_id);
```

Correlation name – variable S in the outer query

Use of "not exists" Clause

- Find all students who have taken all courses offered in the Biology department.

```
select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
                    from course
                    where dept_name = 'Biology')
                   except
                    (select T.course_id
                     from takes as T
                     where S.ID = T.ID));
```

  - First nested query lists all courses offered in Biology
  - Second nested query lists all courses a particular student took

- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$

Test for Absence of Duplicate

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- The **unique** construct evaluates to “true” if a given subquery contains no duplicates .
- Find all courses that were offered at most once in 2017

```
select T.course_id
 from course as T
 where unique ( select R.course_id
                from section as R
                where T.course_id= R.course_id
                      and R.year = 2017);
```

Subqueries in the Form Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

```
select dept_name, avg_salary
from ( select dept_name, avg (salary) as
avg_salary
        from instructor
        group by dept_name)
where avg_salary > 42000;
```

- Note that we do not need to use the **having** clause
- Another way to write above query

With Clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

```
with max_budget (value) as
        (select max(budget)
         from department)
select department.name
from department, max_budget
where department.budget = max_budget.value;
```

Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept_total (dept_name, value) as
        (select dept_name, sum(salary)
         from instructor
         group by dept_name),
    dept_total_avg(value) as
        (select avg(value)
        from dept_total)
    select dept_name
    from dept_total, dept_total_avg
    where dept_total.value > dept_total_avg.value;
```

Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- List all departments along with the number of instructors in each department

```
select dept_name,
        ( select count(*)
          from instructor
          where department.dept_name =
instructor.dept_name)
        as num_instructors
from department;
```

- Runtime error if subquery returns more than one result tuple