

*W4111 – Introduction to Databases
Section 002, Fall 2021*

Lecture 3: ER, Relational, SQL (II)



Contents

Contents

- Course updates and information.
- Covering material (out of planned lecture order) necessary for HW 1.
- ER modeling, notation and clarifying some concepts.
- A little more relational algebra and the dreaded “RelaX Calculator.”
- Diving deeper into SQL and applications.

Course Information and Updates

Course Information and Updates

- There seems to be unprecedeted demand for CS course.
 - This section of W4111's enrollment cap has gone from:
 - 200 to
 - 250 to
 - 320
 - The department approved and enrolled approx. 115 students in the class in the past seven days.
- This section of W4111 **is not a hybrid** class → Guidance was do not record lectures.
 - But, we then received guidance to accommodate students **with reasons** for missing class, e.g. COVID diagnosis, quarantine,
 - I do not believe that I should be determining valid reasons for absence.
- Impact:
 - I am waiving HW 0 submission.
 - I am delaying HW 1 to 29-Sep. This will reduce time between following homework assignments.
 - I will record lectures on Zoom and make available to all students. I did not record lecture 1.

Covering Material Required for HW 1 (Out of syllabus planned order)

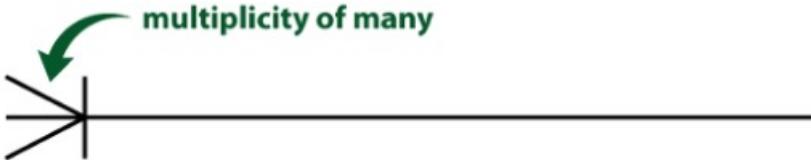
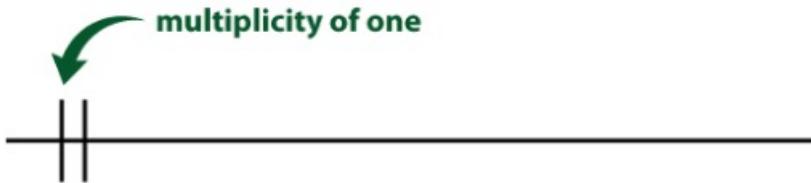
ER Modeling

Crow's Foot Notation – Relationships

- There are several aspects to entity-entity relationships. HW 1 requires understanding two aspects:
 - Cardinality
 - Mandatory/Optional (also known as partial participation, total participation)
- There is a great on-line tutorial if you need to refresh your memory and get additional information.
<https://www.vertabelo.com/blog/crow-s-foot-notation/>
- There are several other notations.

Crow's Foot Notation – Relationships

- The first one (often called multiplicity) refers to the *maximum* number of times that an instance of one entity can be associated with instances in the related entity. It can be one or many.



- The second describes the *minimum* number of times one instance can be related to others. It can be zero or one, and accordingly describes the relationship as optional or mandatory.

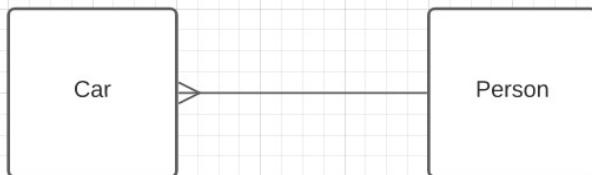


Crow's Foot Notation – Relationships

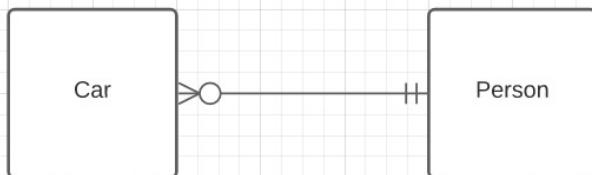
ER Modeling is often progressive definition of details.



- There is a relationship between car and person.
- But, I do not know much about it.

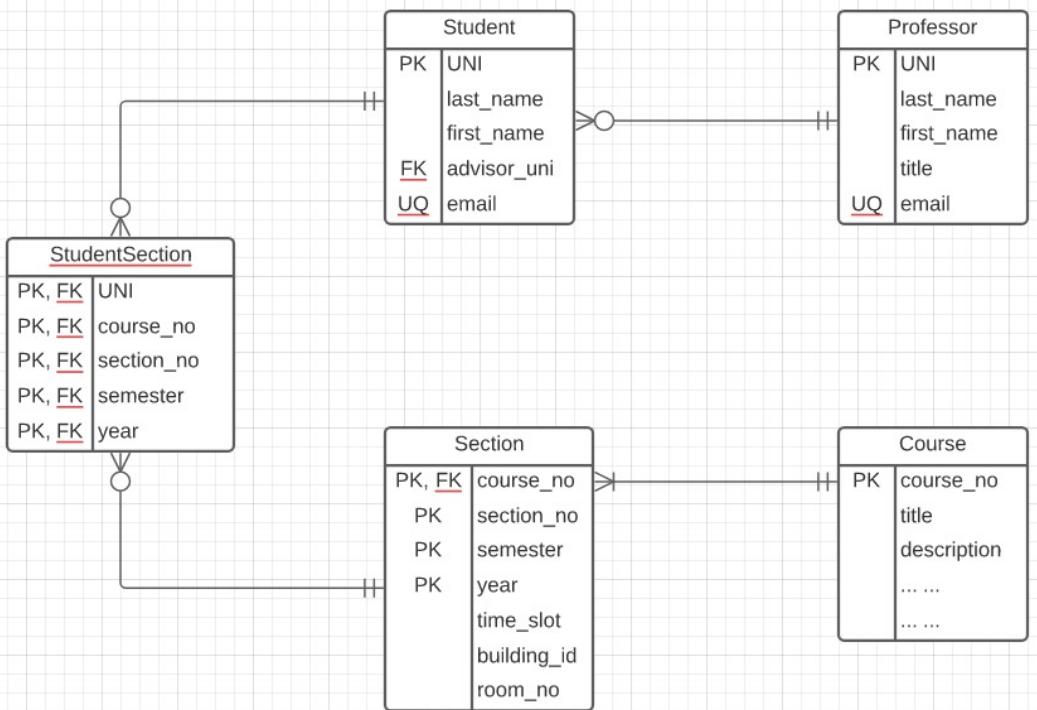


- There is a relationship between car and person.
- A person may have multiple cars, but I do not know if it is required.



- A car is related to exactly one person.
- A person may 0, one or many cars.

Some Details



- **Student-Professor**
 - A student has exactly one advisor.
 - A professor advises 0, 1 or many students.
 - *advisor_uni* is the implementing foreign key.
 - **Section-Course**
 - A course has at least 1 and perhaps many sections.
 - A section is a section of exactly one course.
 - *course_no* is the implementing foreign key.
 - **Student – Section**
 - Shows logical many-to-many, but
 - There is no way to do with foreign keys.
 - One row would need
 - “foreign keys” to multiple rows.
 - This requires an *Associative Entity*.
- Crow's Foot is not a rigorous, formal standard.
- Is an agreed pattern or set of conventions.
 - This diagram shows how I apply the conventions, with extensions.

Relational Algebra

I covered the required material in lecture 2.

SQL

*I covered some concepts in lecture 2.
I will cover additional ones here.*

Dates

Basic components of a calendar date for the most common calendar systems:

D – day

M – month

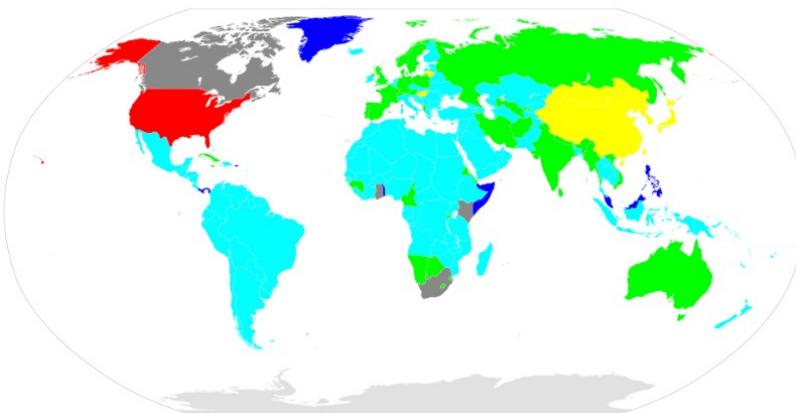
Y – year

Order of the basic components:

B – **big-endian** (year, month, day), e.g. *2016-04-22 or 2016.04.22 or 2016/04/22 or 2016 April 22*

L – **little-endian** (day, month, year), e.g. *22.04.2016 22-04-2016 or 22 April 2016*

M – **middle-endian** (month, day, year), e.g. *04/22/2016 or April 22, 2016*



https://en.wikipedia.org/wiki/Date_format_by_country

Specific formats for the basic components:

yy – two-digit year, e.g. *16*

yyyy – four-digit year, e.g. *2016*

m – one-digit month for months below 10, e.g. *4*

mm – two-digit month, e.g. *04*

mmm – three-letter abbreviation for month, e.g. *Apr*

mmmm – month spelled out in full, e.g. *April*

d – one-digit day of the month for days below 10, e.g. *2*

dd – two-digit day of the month, e.g. *02*

ddd – three-letter abbreviation for day of the week, e.g. *Sat*

dddd – day of the week spelled out in full, e.g. *Saturday*

Separators of the components:

 – oblique stroke (slash)

 – full stop, dot or point (period)

 – hyphen (dash)

 – space

- There are about 7 “standards” in the world.
- And a lot of individual preferences.

Some Interesting Things in “Project”

with

```
basic_view_info as
  (select
    dayofweek(Date(str_to_date(`timestamp`, "%c/%e/%y %H:%i"))) as dow,
    case
      when dayofweek(Date(str_to_date(`timestamp`, "%c/%e/%y %H:%i"))) = 1 then 'Sunday'
      when dayofweek(Date(str_to_date(`timestamp`, "%c/%e/%y %H:%i"))) = 2 then 'Monday'
      when dayofweek(Date(str_to_date(`timestamp`, "%c/%e/%y %H:%i"))) = 3 then 'Tuesday'
      when dayofweek(Date(str_to_date(`timestamp`, "%c/%e/%y %H:%i"))) = 4 then 'Wednesday'
      when dayofweek(Date(str_to_date(`timestamp`, "%c/%e/%y %H:%i"))) = 5 then 'Thursday'
      when dayofweek(Date(str_to_date(`timestamp`, "%c/%e/%y %H:%i"))) = 6 then 'Friday'
      when dayofweek(Date(str_to_date(`timestamp`, "%c/%e/%y %H:%i"))) = 7 then 'Saturday'
    end
    as view_day,
    `minutes delivered` as view_minutes
  from
    some_video_views),
rolled_up as
  (select
    dow, view_day, round(sum(view_minutes), 2) as total_view_minutes
  from
    basic_view_info
  group by dow, view_day
  order by dow)
select view_day, total_view_minutes from rolled_up;
```

- basic_view_info and rolled_up are common table expressions (CTE)
- CTEs
 - Make writing complex queries much easier
 - By breaking the large, complex query into
 - A set of manageable, independently implemented and tested queries.
- You also see some other interesting stuff:
 - CASE statement.
 - Complex operations on columns to produce derived columns.

Pandas

Pandas

“pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

... ...

- DataFrame object for data manipulation with integrated indexing.
- Tools for reading and writing data between in-memory data structures and different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of data sets.
- Label-based slicing, fancy indexing, and subsetting of large data sets.
- Data structure column insertion and deletion.
- Group by engine allowing split-apply-combine operations on data sets.
- Data set merging and joining.
- Hierarchical axis indexing to work with high-dimensional data in a lower-dimensional data structure.
- Time series-functionality: Date range generation[6] and frequency conversions, moving window statistics, moving window linear regressions, date shifting and lagging.
- Provides data filtration.”

[https://en.wikipedia.org/wiki/Pandas_\(software\)](https://en.wikipedia.org/wiki/Pandas_(software))

Pandas

A `Series` is essentially a column, and a `DataFrame` is a multi-dimensional table made up of a collection of Series.

The diagram illustrates the creation of a DataFrame from two Series. On the left, there are two separate tables labeled "Series". The first series has columns "apples" (values: 3, 2, 0, 1) and "oranges" (values: 0, 0, 3, 7). The second series also has columns "apples" (values: 0, 1, 2, 3) and "oranges" (values: 0, 0, 3, 2). A plus sign (+) is placed between the two series, followed by an equals sign (=), indicating the resulting DataFrame. The resulting DataFrame has four rows and two columns, with columns labeled "apples" and "oranges". The "apples" column contains values 3, 2, 0, 1 and the "oranges" column contains values 0, 0, 3, 7.

Series		Series		DataFrame
apples	oranges	apples	oranges	
0	3	0	0	
1	2	1	3	
2	0	2	7	
3	1	3	2	

DataFrames and Series are quite similar in that many operations that you can do with one you can do with the other, such as filling in null values and calculating the mean.

- Very Excel/Google Sheet like, but with
 - Integration with Python and libraries.
 - Some very powerful features.
- Also has basic SQL like functions.
- The ability to read from and right to many data sources.
- We will periodically cover details.

Pandas

`read_csv()`

`read_clipboard()`

`read_excel()`

`read_html()`

`read_json()`

`read_pickle()`

`read_sql()`

`read_table()`



`to_csv()`

`to_clipboard()`

`to_excel()`

`to_html()`

`to_json()`

`to_pickle()`

`to_sql()`

`to_string()`

`to_dict()`

<https://www.plus2net.com/python/pandas-input-output.php>

Switch Back to Notebook

Some Practical ER Modeling and DDL

Worked Example

- Scenario
 - Course with complex course ID.
 - Section
 - Instructor
 - Department

- Do ER (live) diagram in Lucidchart.

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



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

$$\Pi_{course_id} (\sigma_{semester='Fall' \wedge year=2017}(section)) \cup$$
$$\Pi_{course_id} (\sigma_{semester='Spring' \wedge year=2018}(section))$$

(a,b,c,d)

(x,y)

Project(a,



Union Operation (Cont.)

- Result of:

$$\Pi_{course_id} (\sigma_{semester="Fall"} \wedge year=2017 (section)) \cup$$
$$\Pi_{course_id} (\sigma_{semester="Spring"} \wedge year=2018 (section))$$

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.

$$\begin{aligned}\prod_{course_id} (\sigma_{semester='Fall'} \wedge year=2017(section)) \cap \\ \prod_{course_id} (\sigma_{semester='Spring'} \wedge year=2018(section))\end{aligned}$$

- 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

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

<i>course_id</i>
CS-347
PHY-101

Same “arity”

Select DB (W4111 SimpleUnion) ▾

students

- `id` string
- `first_name` string
- `last_name` string
- `email` string
- `year` string

faculty

- `id` string
- `first_name` string
- `last_name` string
- `email` string
- `title` string
- `hire_date` string

- 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 combining them does not make sense.
- You can shape two incompatible tables using *project operations*. For example
 - $\pi \text{first_name}, \text{last_name}, \text{email} (\text{students})$
 \cap
 $\pi \text{first_name}, \text{last_name}, \text{email} (\text{faculty})$
 - $\pi \text{last_name}, \text{email}, \text{title} \leftarrow \text{'Student'} (\text{students})$
 \cup
 $\pi \text{last_name}, \text{email}, \text{title} (\text{faculty})$



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 \leftarrow and works like assignment in a programming language.
- Example: Find all instructor in the “Physics” and Music department.

$$\text{Physics} \leftarrow \sigma_{\text{dept_name} = \text{“Physics”}}(\text{instructor})$$
$$\text{Music} \leftarrow \sigma_{\text{dept_name} = \text{“Music”}}(\text{instructor})$$
$$\text{Physics} \cup \text{Music}$$

- 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:

$$\rho_{x(A_1, A_2, \dots, A_n)}(E)$$

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

$$\sigma_{dept_name = "Physics"} \wedge salary > 90,000 (instructor)$$

- Query 2
- $$\sigma_{dept_name = "Physics"} (\sigma_{salary > 90.000} (instructor))$$
- 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

$$\sigma_{dept_name = "Physics"}(instructor \bowtie_{instructor.ID = teaches.ID} teaches)$$

- Query 2
- $$(\sigma_{dept_name = "Physics"}(instructor)) \bowtie_{instructor.ID = teaches.ID} teaches$$
- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.

Some More SQL

Data Types and Functions



Domain Types in SQL

- **char(*n*)**. Fixed length character string, with user-specified length *n*.
- **varchar(*n*)**. Variable length character strings, with user-specified maximum length *n*.
- **int**. Integer (a finite subset of the integers that is machine-dependent).
- **smallint**. Small integer (a machine-dependent subset of the integer domain type).
- **numeric(*p,d*)**. Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point. (ex., **numeric(3,1)**, allows 44.5 to be stored exactly, but not 444.5 or 0.32)
- **real, double precision**. Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(*n*)**. Floating point number, with user-specified precision of at least *n* digits.
- More are covered in Chapter 4.



Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
 - Example: `date '2005-7-27'`
- **time:** Time of day, in hours, minutes and seconds.
 - Example: `time '09:00:30'` `time '09:00:30.75'`
- **timestamp:** date plus time of day
 - Example: `timestamp '2005-7-27 09:00:30.75'`
- **interval:** period of time
 - Example: `interval '1' day`
 - Subtracting a date/time/timestamp value from another gives an interval value
 - Interval values can be added to date/time/timestamp values

Note:

- All implementations of SQL support a common core, but ...
- They have proprietary extensions: operators on types, additional types.



Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
 - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
 - **clob**: character large object -- object is a large collection of character data
- When a query returns a large object, a pointer is returned rather than the large object itself.

Note:

- CLOBs and BLOBs are very uncommon.
- Today applications keep documents, images, CAD files, etc. in “block/object storage.”
- The database contains just the hyperlinks to the files/content.
- We will see some examples when we talk about web applications.

MySQL Cheat Sheet

<https://websitesetup.org/wp-content/uploads/2020/04/MySQL-Cheat-Sheet-websitesetup.org.pdf>

Pretty good, concise place to find information about MySQL.

<https://dev.mysql.com/doc/refman/8.0/en/data-types.html>

Data Types

Data types indicate what type of information you can store in a particular column of your table.

MySQL has three main categories of data types:

- Numeric
- Text
- Date/time

These lists are
not complete.

Blob and Text Data Types

BLOB binary range enables you to store larger amounts of text data. The maximum length of a BLOB is **65,535 ($2^{16} - 1$) bytes**. BLOB values are stored using a 2-byte length prefix.

NB: Since text data can get long, always double-check that you do not exceed the maximum lengths. The system will typically generate a warning if you go beyond the limit. But if nonspace characters get truncated, you may just receive an error without a warning.

- **TINYBLOB** – sets the maximum column length at 255 ($2^8 - 1$) bytes. TINYBLOB values are stored using a 1-byte length prefix.
- **MEDIUMBLOB** – sets the maximum column length at 16,777,215 ($2^{24} - 1$) bytes. MEDIUMBLOB values are stored using a 3-byte length prefix.
- **LONGBLOB** – sets the maximum column length at 4,294,967,295 or 4GB ($2^{32} - 1$) bytes. LONGBLOB values are stored using a 4-byte length prefix.

Note: The max length will also depend on the maximum packet size that you configure in the client/server protocol, plus available memory.

If unsigned, the column will expand to hold the data up till a certain upper boundary range.

- **BIT([M])** — specify a bit-value type. **M** stands for the number of bits per value, ranging from 1 to 64. The default is 1 if no **T** specified.
- **ZEROFILL** — auto-add UNSIGNED attribute to the column. Deprecated since the MySQL 8.0.17 version.
- **TINYINT(M)** — the smallest integer with a range of -128 to 127.
 - **TINYINT(M) [UNSIGNED]** — the range is 0 to 255.
 - **BOOL, BOOLEAN** — synonyms for TINYINT(1)
- **SMALLINT(M)** — small integer with a range of -32768 and 32767.
 - **SMALLINT(M) [UNSIGNED]** — the range is 0 to 65535.
- **MEDIUMINT(M)** — medium integer with a range of -8388608 to 8388607.
 - **MEDIUMINT(M) [UNSIGNED]** — the range is 0 to 16777215.
- **INT(M) and INTEGER (M)** — normal range integer with a range of -2147483648 to 2147483647.
 - **INT(M)[UNSIGNED] and INTEGER (M)[UNSIGNED]** — the range is 0 to 4294967295.
- **BIGINT(M)** — the largest integer with a range of -9223372036854775808 to 9223372036854775807.
 - **BIGINT(M) [UNSIGNED]** — the range is 0 to 8446744073709551615.
- **DECIMAL (M, D)** — store a double value as a string. **M** specifies the total number of digits. **D** stands for the number of digits after the decimal point. Handy for storing currency values.
 - Max number of M is 65. If omitted, the default M value is 10.
 - Max number of D is 30. If omitted, the default D is 0.
- **FLOAT (M, D)** — record an approximate number with a floating decimal point. The support for FLOAT is removed as of MySQL 8.0.17 and above.
 - Permissible values ranges are -3.402823466E+38 to -1.175494351E-38, 0, and 1.175494351E-38 to 3.402823466E+38.

MySQL Cheat Sheet

<https://websitesetup.org/wp-content/uploads/2020/04/MySQL-Cheat-Sheet-websitesetup.org.pdf>

Pretty good, concise place to find information about MySQL.

<https://dev.mysql.com/doc/refman/8.0/en/data-types.html>

Text Storage Formats

- **CHAR** — specifies the max number of non-binary characters you can store. The range is from 0 to 255.
- **VARCHAR** — store variable-length non-binary strings. The maximum number of characters you can store is 65,535 (equal to the max row size).
 - VARCHAR values are stored as a 1-byte or 2-byte length prefix plus data, unlike CHAR values.
- **BYNARY** — store binary data in the form of byte strings. Similar to CHAR.
- **VARBYNARY** — store binary data of variable length in the form of byte strings. Similar to VARCHAR.
- **ENUM** — store permitted text values that you enumerated in the column specification when creating a table.
 - ENUM columns can contain a maximum of 65,535 distinct elements and have > 255 unique element list definitions among its ENUM.
- **SET** — another way to store several text values that were chosen from a predefined list of values.
 - SET column can contain a maximum of 64 distinct members and have > 255 unique element list definitions among its SET.

These lists are not complete for the basic groups, and there are some advanced types (JSON, Spatial). You learn all of this by practicing, trial and error, etc.

Date and Time Data Types

As the name implies, this data type lets you store the time data in different formats.

- **DATE** — use it for values with a date part only. MySQL displays DATE values in the 'YYYY-MM-DD' format.
 - Supported data range is '1000-01-01' to '9999-12-31'.
- **DATETIME** — record values that have both date and time parts. The display format is 'YYYY-MM-DD hh:mm:ss'.
 - Supported data range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'.
- **TIMESTAMP** — add more precision to record values that have both date and time parts, up till microseconds in UTC.
 - Supported data range is '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC.
- **TIME** — record just time values in either 'hh:mm:ss' or 'hh:mm:ss' format. The latter can represent elapsed time and time intervals.
 - Supported data range is '-838:59:59' to '838:59:59'.
- **YEAR** — use this 1-byte type used to store year values.
 - A 4-digit format displays YEAR values as 0000, with a range between 1901 to 2155.
 - A 2-digit format displays YEAR values as 00. The accepted range is '0' to '99' and MySQL will convert YEAR values in the ranges 2000 to 2069 and 1970 to 1999.



String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator **like** uses patterns that are described using two special characters:
 - percent (%). The % character matches any substring.
 - underscore (_). The _ character matches any character.
- Find the names of all instructors whose name includes the substring “dar”.

```
select name  
from instructor  
where name like '%dar%'
```

- Match the string “100%”

```
like '100 \%' escape '\'
```

in that above we use backslash (\) as the escape character.



String Operations (Cont.)

- Patterns are case sensitive.
- Pattern matching examples:
 - 'Intro%' matches any string beginning with “Intro”.
 - '%Comp%' matches any string containing “Comp” as a substring.
 - '___' matches any string of exactly three characters.
 - '___ %' matches any string of at least three characters.
- SQL supports a variety of string operations such as
 - concatenation (using “||”)
 - converting from upper to lower case (and vice versa)
 - finding string length, extracting substrings, etc.

Operations and Functions

MySQL Type Conversion

BINARY 'string'
CAST (expression AS datatype)
CONVERT (expression, datatype)

MySQL Grouping Functions

AVG	MAX
BIT_AND	STD
BIT_OR	STDDEV
COUNT	SUM
GROUP_CONCAT	VARIANCE
MIN	

MySQL Mathematical Functions

ABS	COS
SIGN	SIN
MOD	TAN
FLOOR	ACOS
CEILING	ASIN
ROUND	ATAN, ATAN2
DIV	COT
EXP	RAND
LN	LEAST
LOG, LOG2, LOG10	GREATEST
POW	DEGREES
POWER	RADIANS
SQRT	TRUNCATE
PI	

MySQL String Functions

ASCII	SUBSTRING
ORD	MID
CONV	SUBSTRING_INDEX
BIN	LTRIM
OCT	RTRIM
HEX	TRIM
CHAR	SOUNDEX
CONCAT	SPACE
CONCAT_WS	REPLACE
LENGTH	REPEAT
CHAR_LENGTH	REVERSE
BIT_LENGTH	INSERT
LOCATE	ELT
INSTR	FIELD
LPAD	LCASE
RPAD	UCASE
LEFT	LOAD_FILE
RIGHT	QUOTE

MySQL Date and Time Functions

DAYOFWEEK	DATE_SUB
WEEKDAY	ADDDATE
DAYOFMONTH	SUBDATE
DAYOFYEAR	EXTRACT
MONTH	TO_DAYS
DAYNAME	FROM_DAYS
MONTHNAME	DATE_FORMAT
QUARTER	TIME_FORMAT
WEEK	CURRENT_DATE
YEAR	CURRENT_TIME
YEARWEEK	NOW
HOUR	SYSDATE
MINUTE	UNIX_TIMESTAMP
SECOND	FROM_UNIXTIME
PERIOD_ADD	SEC_TO_TIME
PERIOD_DIFF	TIME_TO_SEC
DATE_ADD	

MySQL Control Flow Functions

IF	NULLIF
IFNULL	

Almost all programming languages have similar function libraries.

Operations and Functions

- How do you learn all of these functions?
- Well, “How do I get to Carnegie Hall?”
- “Practice.”

- Switch to notebook.

Summary

- SQL Data Types:
 - There is a common, standard core set of data types that all SQL implementations support.
 - Most SQL implementations have additional types and extensions.
- Functions on Data Types:
 - There is a common, standard core set of functions for each data type that all SQL implementations support.
 - Most SQL implementations have additional functions and extensions.
- I do not expect you to memorize the types, functions, etc.
You can look them up, play with them, ... On homework assignments and exams.
- We saw some interesting additional features that we will cover later in this lecture:
 - GROUP BY
 - ORDER BY

Order By



Ordering the Display of Tuples

- List in alphabetic order the names of all instructors

```
select distinct name  
from instructor  
order by name
```

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
 - Example: **order by name desc**
- Can sort on multiple attributes
 - Example: **order by dept_name, name**

Show notebook for order by example.

INSERT, UPDATE, DELETE



Updates to tables

- **Insert**
 - `insert into instructor values ('10211', 'Smith', 'Biology', 66000);`
- **Delete**
 - Remove all tuples from the *student* relation
 - `delete from student`
- **Drop Table**
 - `drop table r`
- **Alter**
 - `alter table r add A D`
 - where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A*.
 - All existing tuples in the relation are assigned *null* as the value for the new attribute.
 - `alter table r drop A`
 - where *A* is the name of an attribute of relation *r*
 - Dropping of attributes not supported by many databases.



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.

Aggregate Functions



Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value

min: minimum value

max: maximum value

sum: sum of values

count: number of values

Note: Some database implementations have additional aggregate functions.



Aggregate Functions – Group By

- Find the average salary of instructors in each department
 - `select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name;`

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

<i>dept_name</i>	<i>avg_salary</i>
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Another View

Employees

DEPARTMENT_ID	SALARY
10	5500
20	15000
20	7000
30	12000
30	5100
30	4900
30	5800
30	5600
40	7500
40	8000
50	9000
50	8500
50	9500
50	8500
50	10500
50	10000
50	9500

Sum of Salary in Employees table for each department

DEPARTMENT_ID	SUM(SALARY)
10	5500
20	22000
30	33400
40	15500
50	65550

- GROUP BY column list
 - Forms partitions containing multiple rows.
 - All rows in a partition have the same values for the GROUP BY columns.
- The aggregate functions
 - Merge the non-group by attributes, which may differ from row to row.
 - Into a single value for each attribute.
- The result is one row per distinct set of GROUP BY values.
- There may be multiple non-GROUP BY COLUMNS, each with its own aggregate function.
- You can use HAVING in place of WHERE on the GROUP BY result.



Aggregate Functions Examples

- Find the average salary of instructors in the Computer Science department
 - **select avg (salary)
from instructor
where dept_name= 'Comp. Sci.';**
- Find the total number of instructors who teach a course in the Spring 2018 semester
 - **select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2018;**
- Find the number of tuples in the *course* relation
 - **select count (*)
from course;**



Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list

- /* erroneous query */
select *dept_name*, *ID*, **avg** (*salary*)
from *instructor*
group by *dept_name*;



Aggregate Functions – Having Clause

- Find the names and average salaries of all departments whose average salary is greater than 42000

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

- Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

JOIN



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 A1, A2, ... An
from r1 natural join r2 natural join .. natural join rn
where P;
```



Student Relation

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>
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

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
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	<i>null</i>



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.)

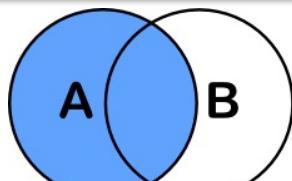
- The **on** condition allows a general predicate over the relations being joined.
- This predicate is written like a **where** clause predicate except for the use of the keyword **on**.
- Query example

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

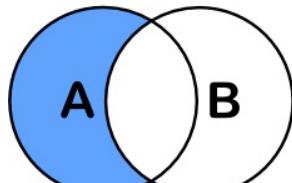
- The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.
- Equivalent to:

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

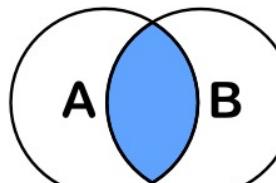
One Way to Think About Joins



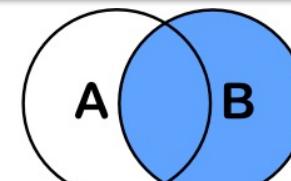
```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
```



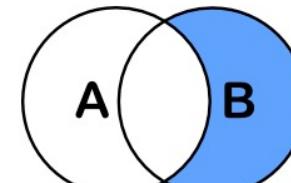
```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
WHERE B.key IS NULL
```



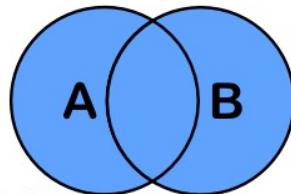
```
SELECT <auswahl>
FROM tabelleA A
INNER JOIN tabelleB B
ON A.key = B.key
```



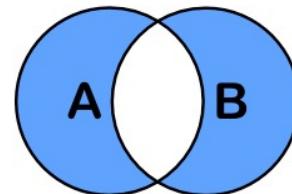
```
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
```



```
SELECT <auswahl>
FROM tabelleA A
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
```

Set Operations

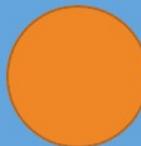
Set Operations

Visual Explanation of UNION, INTERSECT, and EXCEPT operators

Left Query

Right Query

Final Result



UNION



=>



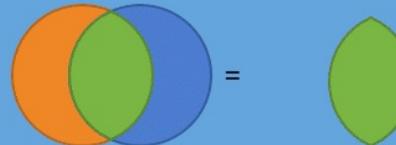
Combine rows from
both queries.



INTERSECT



=>



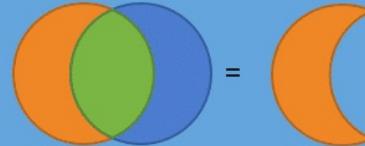
Keep only rows in common to
both queries.



EXCEPT



=>



Keep rows from left query that
aren't included in the right query



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

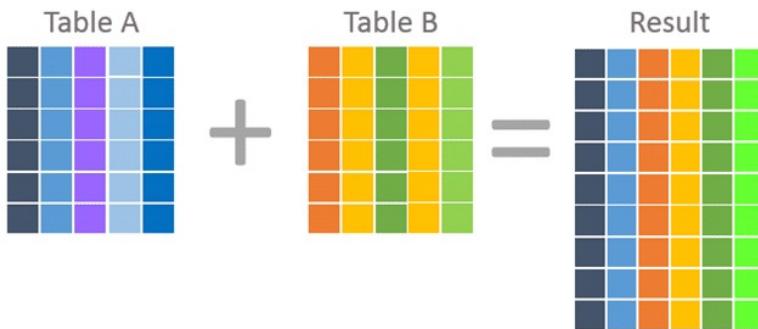
```
(select course_id from section where sem = 'Fall' and year = 2017)
intersect
(select course_id from section where sem = 'Spring' and year = 2018)
```

- 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)
```

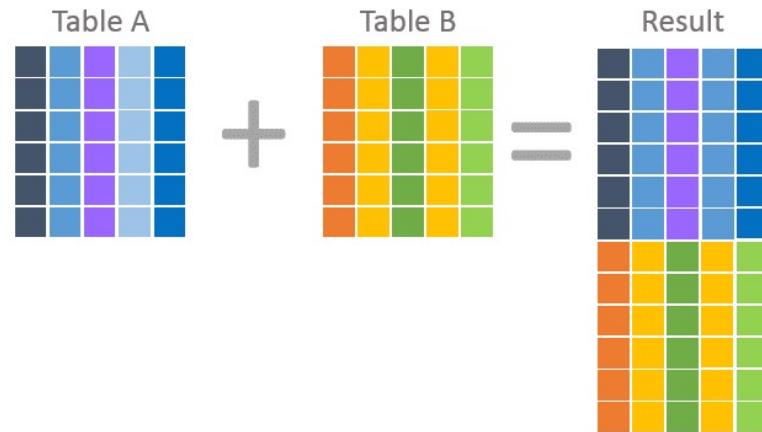
JOIN and UNION – A Final Word

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 other.”

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 A1, A2, ..., An
  from r1, r2, ..., rm
 where P
```

as follows:

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

$B <\text{operation}> (\text{subquery})$

B is an attribute and $<\text{operation}>$ to be defined later.

- **Select clause:**

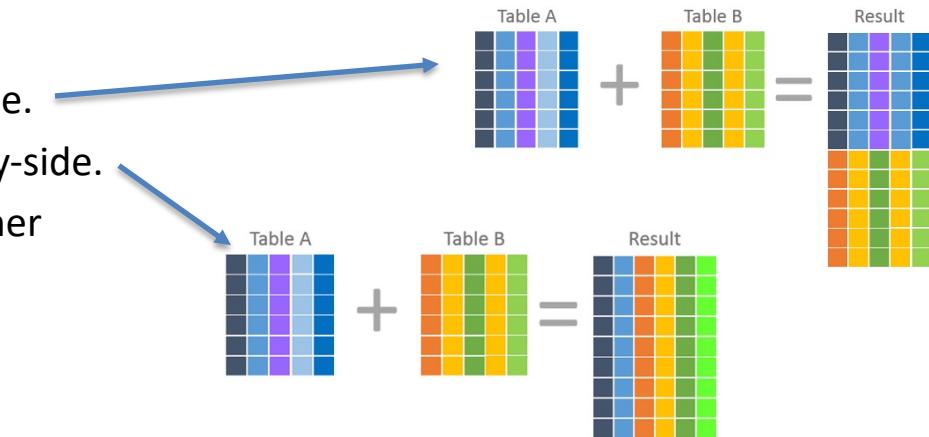
A_i can be replaced be a subquery that generates a single value.

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;

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

- 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

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 A1, A2, ..., An
  from r1, r2, ..., rm
 where P
```

as follows:

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

$B <\text{operation}> (\text{subquery})$

B is an attribute and $<\text{operation}>$ to be defined later.

- **Select clause:**
 A_i can be replaced by a subquery that generates a single value.

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



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);
```

- Note: Above query can be written in a much simpler manner.
The formulation above is simply to illustrate SQL features



Set Comparison



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

- $F <\text{comp}> \text{some } r \Leftrightarrow \exists t \in r \text{ such that } (F <\text{comp}> t)$
Where comp can be: $<$, \leq , $>$, $=$, \neq

$(5 < \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{true}$ (read: 5 < some tuple in the relation)

$(5 < \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{false}$

$(5 = \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$

$(5 \neq \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$ (since $0 \neq 5$)

$(= \text{some}) \equiv \text{in}$

However, $(\neq \text{some}) \not\equiv \text{not in}$



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

- $F <\text{comp}> \text{all } r \Leftrightarrow \forall t \in r (F <\text{comp}> t)$

(5 < all

0
5
6

) = false

(5 < all

6
10

) = true

(5 = all

4
5

) = false

(5 ≠ all

4
6

) = true (since $5 \neq 4$ and $5 \neq 6$)

$(\neq \text{all}) \equiv \text{not in}$

However, $(= \text{all}) \not\equiv \text{in}$



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
- Correlated subquery** – the inner 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 $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using = all and its variants



Test for Absence of Duplicate Tuples

- 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 From Clause



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

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



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