W4111 – Introduction to Databases
Section 002, Fall 2025
Lecture 3: Foundation – ER, Relational, SQL (2)

Contents

Notes and Content

- ER Model/Modeling:
  - Generalization/Specialization
  - Aggregation
- Relational algebra
  - Left Join, Right Join, Semi-Join
  - Division
- SQL
  - Aggregation
  - Subquery review and continuation
  - Complex JOIN
  - Generalization/Specialization
- Project concepts: (Web) applications, REST, data engineering

ER Model/ER Modeling

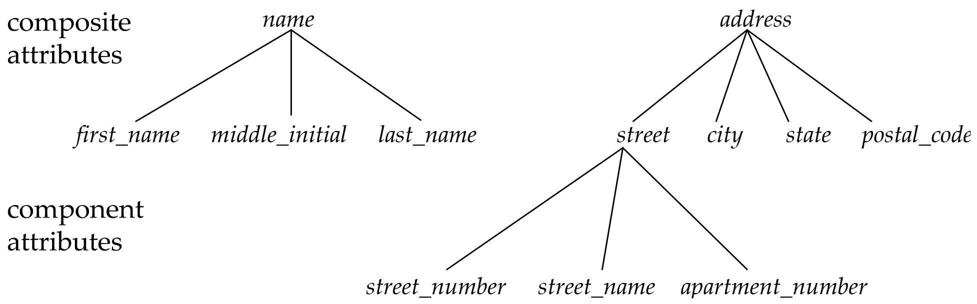
Complex Attributes

Complex Attributes

- Attribute types:
  - **Simple** and **composite** attributes.
  - **Single-valued** and **multivalued** attributes
    - Example: multivalued attribute: *phone_numbers*
  - **Derived** attributes
    - Can be computed from other attributes
    - Example: age, given date_of_birth
- **Domain** – the set of permitted values for each attribute

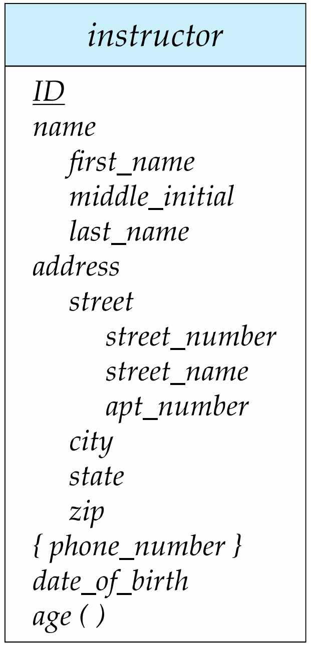
Composite Attributes

- Composite attributes allow us to divided attributes into subparts (other attributes).



Representing Complex Attributes in ER Diagram

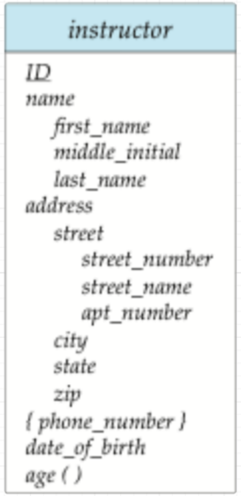
Swith to Lucidchart and show how to model. Show a couple of options/design choices.

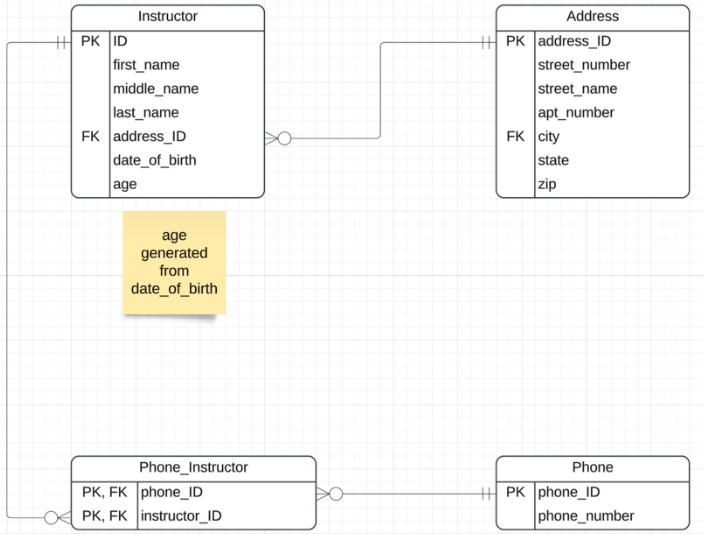


```
instructor
ID
name
    first_name
    middle_initial
    last_name
address
    street
        street_number
        street_name
        apt_number
    city
    state
    zip
{ phone_number }
date_of_birth
age ( )
```

Complex Attributes in Crow's Foot Notation

- Like much ER modeling, there are design options and choices
- Composite
  - Use a 2nd entity type and foreign key relationship.
  - "Explode" into individual attributes.
- Multi-valued
  - Use a 2nd entity type with a foreign key or associative entity.
  - Note: I assume in my design that the relationship between instructor and phone was many-to-many. I would likely also add a "phone_type" property, e.g. home, mobile, … …
- The only way to handle generated is with a note or annotation.





Switch to Notebook

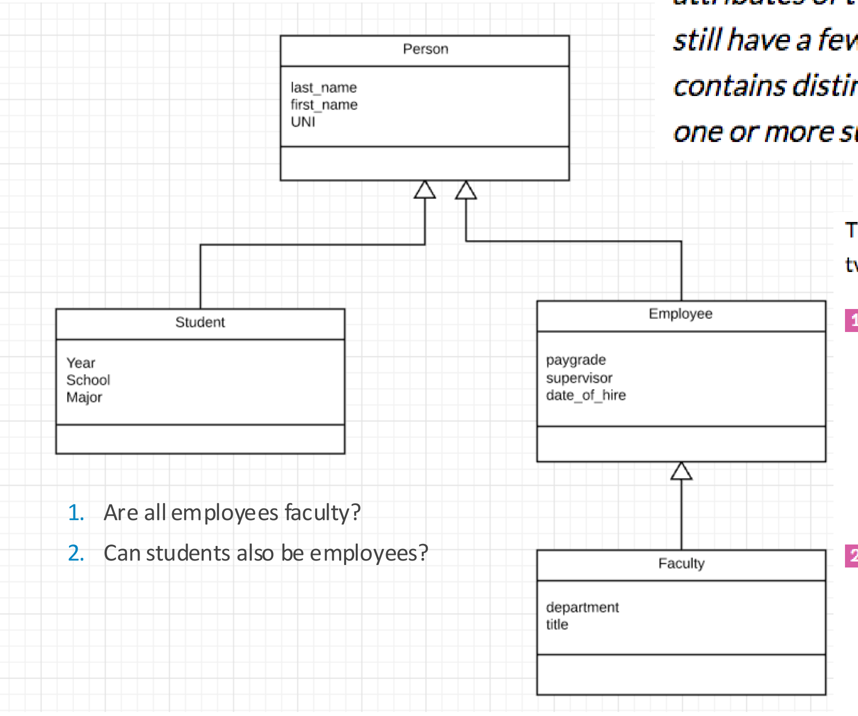
Inheritance/Specialization/Generalization

Specialization

- Top-down design process; we designate sub-groupings within an entity set that are distinctive from other entities in the set.
- These sub-groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
- Depicted by a *triangle* component labeled ISA (e.g., *instructor* "is a" *person*).
- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.

Inheritance/Specialization

In the process of designing our entity relationship diagram for a database, we may find that attributes of two or more entities overlap, meaning that these entities seem very similar but still have a few differences. In this case, we may create a subtype of the parent entity that contains distinct attributes. A parent entity becomes a supertype that has a relationship with one or more subtypes.



1. Are all employees faculty?
2. Can students also be employees?

The subclass association line is labeled with specialization constraints. Constraints are described along two dimensions:

1 **incomplete/complete**

- In an **incomplete** specialization only some instances of the parent class are specialized (have unique attributes). Other instances of the parent class have only the common attributes.
- In a **complete** specialization, every instance of the parent class has one or more unique attributes that are not common to the parent class.

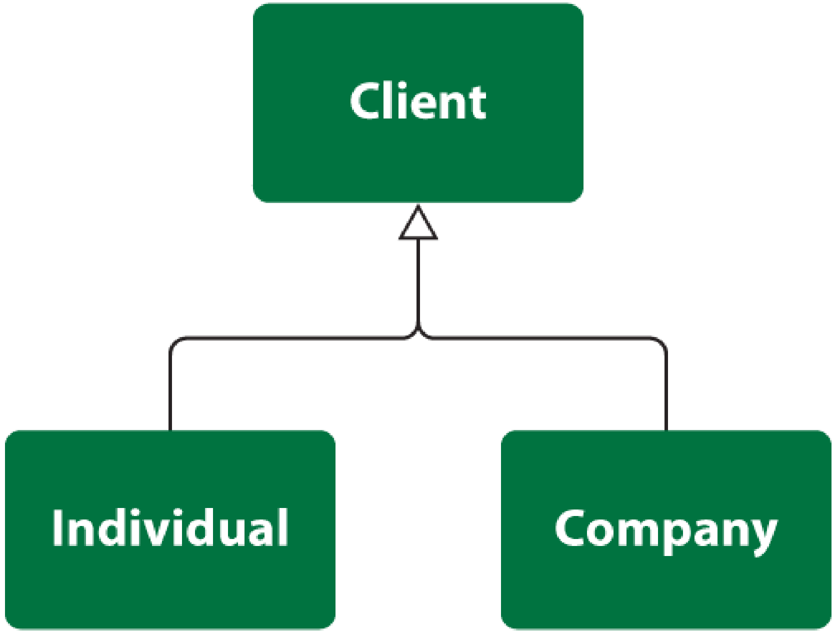
2 **disjoint/overlapping**

- In a **disjoint** specialization, an object could be a member of only one specialized subclass.
- In an **overlapping** specialization, an object could be a member of more than one specialized subclass.

http://www.vertabelo.com/blog/technical-articles/inheritance-in-a-relational-database

Specialization

In class Client we distinguish two subtypes: Individual and Company. This specialization is disjoint (client can be an individual or a company) and complete (these are all possible subtypes for supertype).

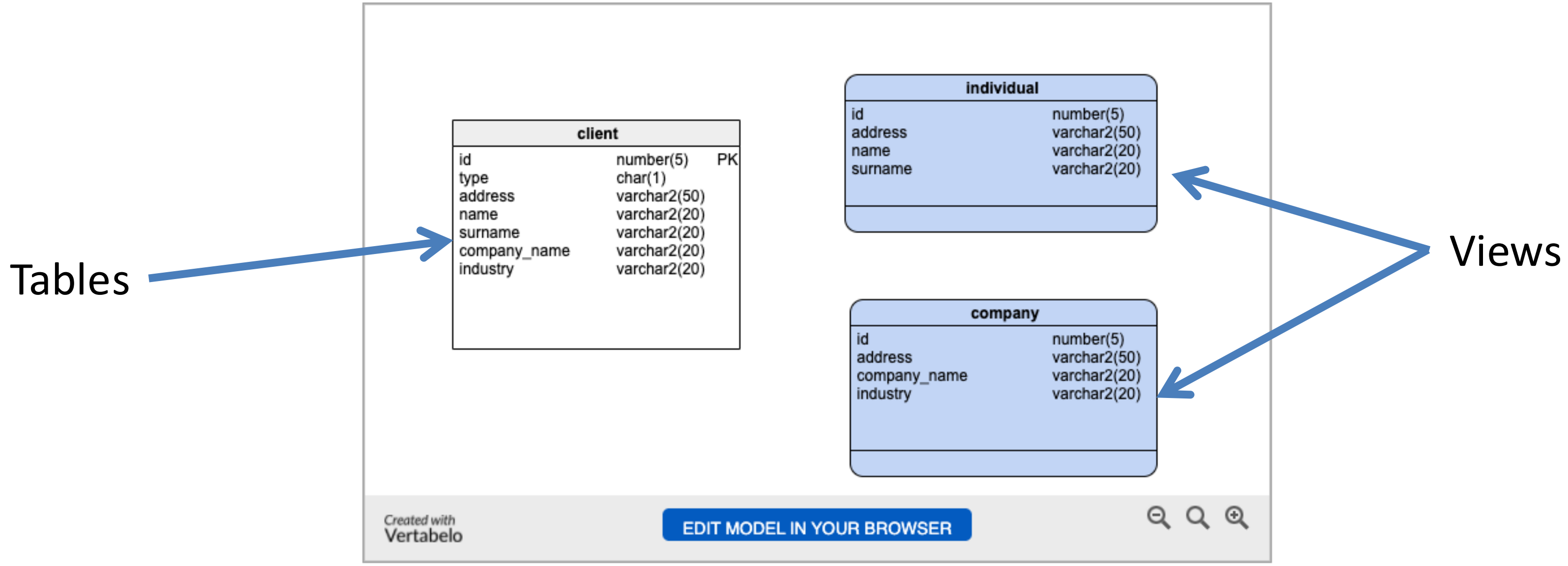


One Table

One table implementation

In a one table implementation, table `client` has attributes of both types.

The diagram below shows the table `client` and two views: `individual` and `company`:



client

id	number(5)	PK
type	char(1)	
address	varchar2(50)	
name	varchar2(20)	
surname	varchar2(20)	
company_name	varchar2(20)	
industry	varchar2(20)	

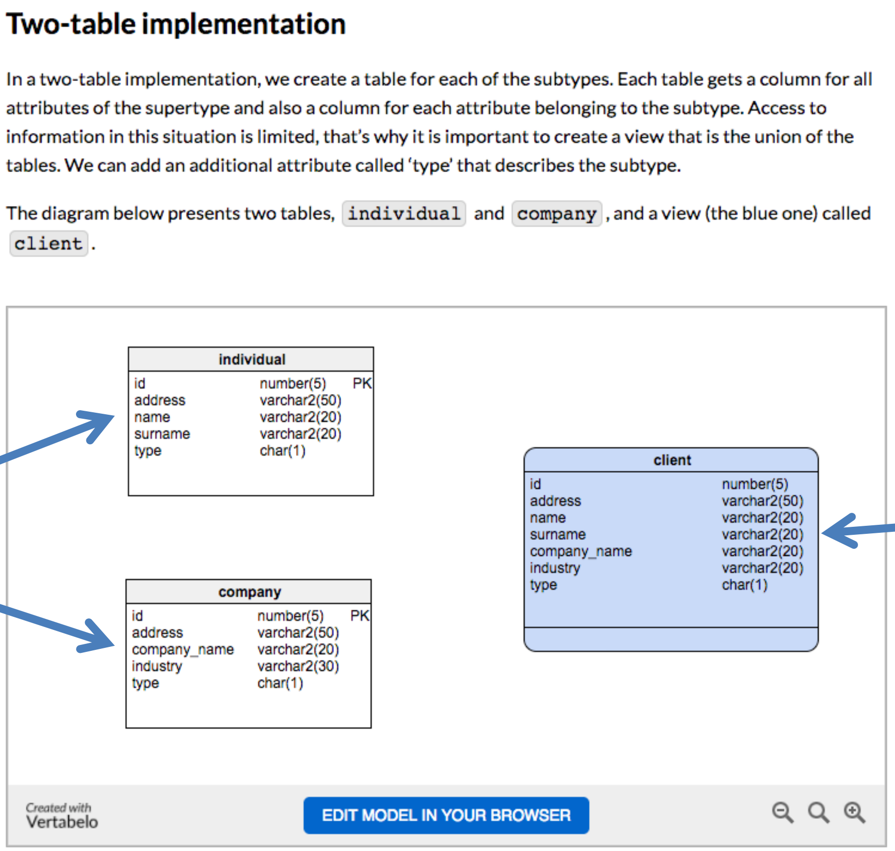
individual

id	number(5)
address	varchar2(50)
name	varchar2(20)
surname	varchar2(20)

company

id	number(5)
address	varchar2(50)
company_name	varchar2(20)
industry	varchar2(20)

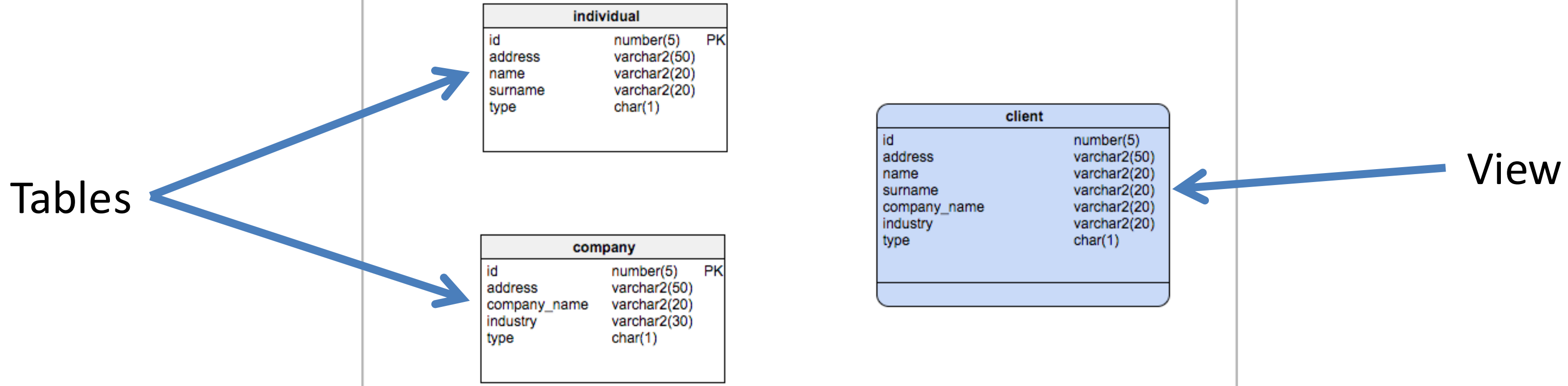
Two Table



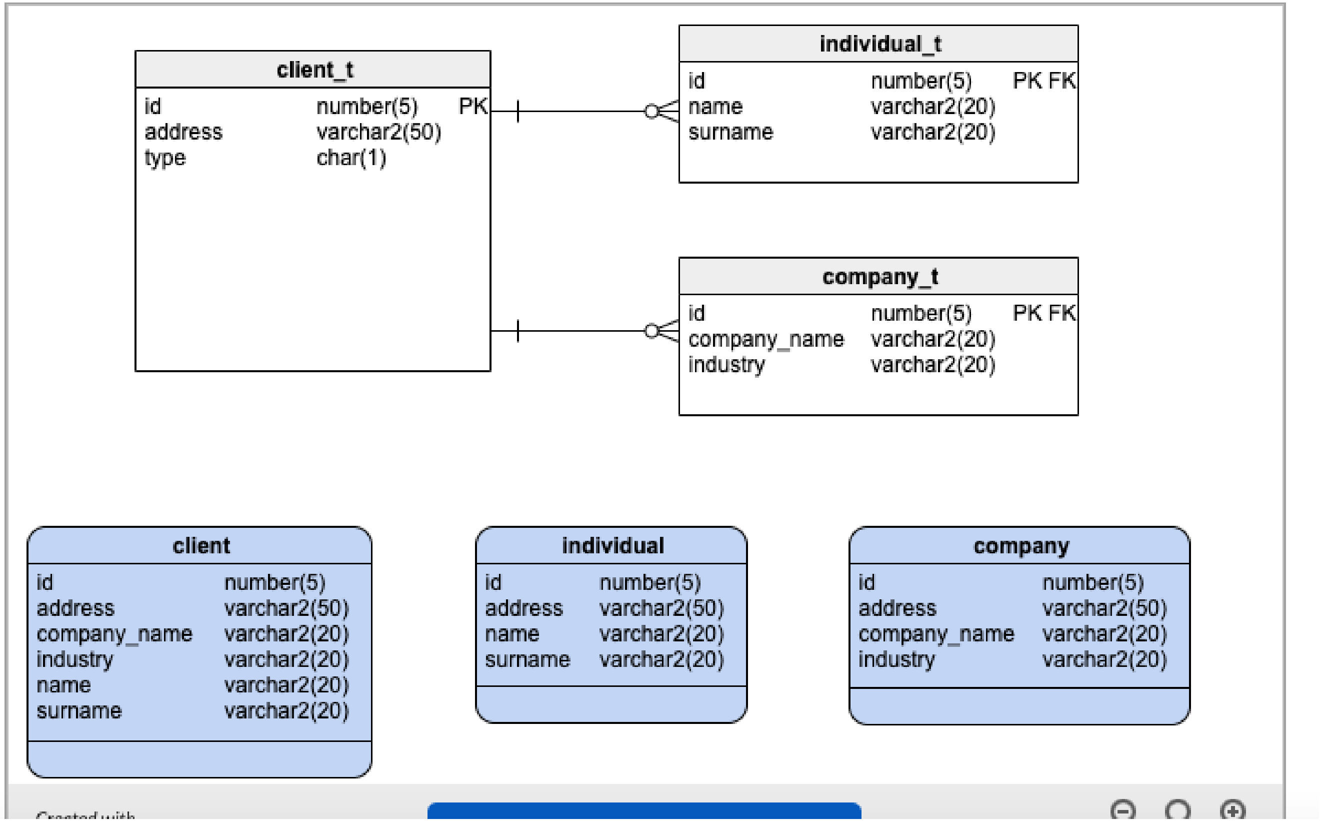
Two-table implementation

In a two-table implementation, we create a table for each of the subtypes. Each table gets a column for all attributes of the supertype and also a column for each attribute belonging to the subtype. Access to information in this situation is limited, that's why it is important to create a view that is the union of the tables. We can add an additional attribute called 'type' that describes the subtype.

The diagram below presents two tables, `individual` and `company`, and a view (the blue one) called `client`.



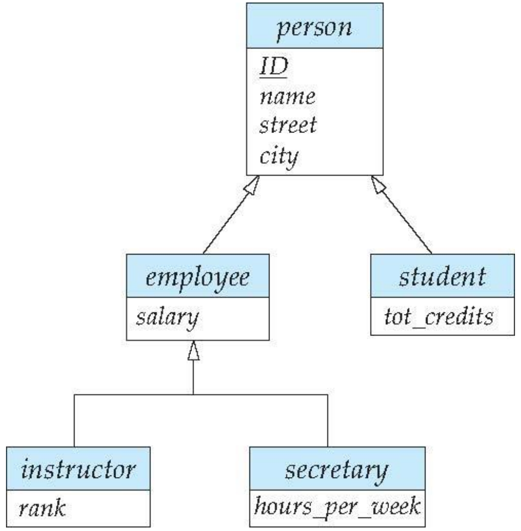
Three Table



client_t
id number(5) PK
address varchar2(50)
type char(1)
individual_t
id number(5) PK FK
name varchar2(20)
surname varchar2(20)
company_t
id number(5) PK FK
company_name varchar2(20)
industry varchar2(20)
client
id number(5)
address varchar2(50)
company_name varchar2(20)
industry varchar2(20)
name varchar2(20)
surname varchar2(20)
individual
id number(5)
address varchar2(50)
name varchar2(20)
surname varchar2(20)
company
id number(5)
address varchar2(50)
company_name varchar2(20)
industry varchar2(20)

Specialization Example

- **Overlapping** – *employee* and *student*
- **Disjoint** – *instructor* and *secretary*
- Total and partial



Representing Specialization via Schemas

- Method 1:
  - Form a schema for the higher-level entity
  - Form a schema for each lower-level entity set, include primary key of higher-level entity set and local attributes

schema	attributes
person	ID, name, street, city
student	ID, tot_cred
employee	ID, salary

  - Drawback: getting information about, an *employee* requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema

Representing Specialization as Schemas (Cont.)

- Method 2:
  - Form a schema for each entity set with all local and inherited attributes

schema	attributes
person	ID, name, street, city
student	ID, name, street, city, tot_cred
employee	ID, name, street, city, salary

  - Drawback: *name, street* and *city* may be stored redundantly for people who are both students and employees

Generalization

- **A bottom-up design process** – combine a number of entity sets that share the same features into a higher-level entity set.
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.
- The terms specialization and generalization are used interchangeably.

Completeness constraint

- **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.
  - **total**: an entity must belong to one of the lower-level entity sets
  - **partial**: an entity need not belong to one of the lower-level entity sets

Completeness constraint (Cont.)

- Partial generalization is the default.
- We can specify total generalization in an ER diagram by adding the keyword **total** in the diagram and drawing a dashed line from the keyword to the corresponding hollow arrow-head to which it applies (for a total generalization), or to the set of hollow arrow-heads to which it applies (for an overlapping generalization).
- The *student* generalization is total: All student entities must be either graduate or undergraduate. Because the higher-level entity set arrived at through generalization is generally composed of only those entities in the lower-level entity sets, the completeness constraint for a generalized higher-level entity set is usually total

Relational Algebra

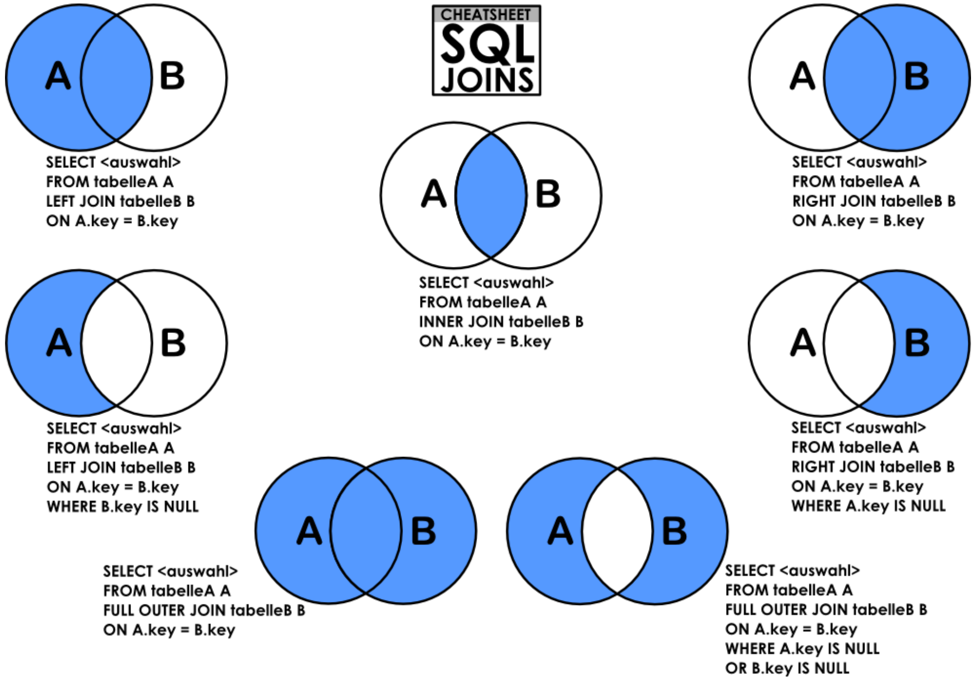
What are all those other Symbols?

τ	order by
γ	group by
$\neg$	negation
$\div$	set division
⋈	natural join, theta-join
⟕	left outer join
⟖	right outer join
⟗	full outer join
⋉	left semi join
⋊	right semi join
▷	anti-join

- Some of the operators are useful and "common," but not always considered part of the core algebra.
- Some of these are pretty obscure
  - Division
  - Anti-Join
  - Left semi-join
  - Right semi-join
- Most SQL engines do not support them.
  - You can implement them using combinations of JOIN, SELECT, WHERE, ...
  - But, I cannot every remember using them in applications I have developed.
- Outer JOIN is very useful, but less common. We will cover.
- There are also some "patterns" or "terms"
  - Equijoin
  - Non-equi join
  - Natural join
  - Theta join
  - ... ...
- I may ask you to define these terms on some exams or the obscure operatorsbecause they may be common internships/job interview questions.

Thinking about JOINs

- Some terms:
  - Natural Join
    - Equality of A and B columns
    - With the same name.
  - Equijoin
    - Explicitly specify columns that must have the same value.
    - A.x=B.z AND A.q=B.w
  - Theta Join: Arbitrary predicate.
- Inner Join
  - JOIN "matches" rows in A and B.
  - Result contains ONLY the matched pairs.
- What I want is:
  - All the rows that matched.
  - And the rows from A that did not match?
  - OUTER JOIN (⟕, ⟖)



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

```
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
```

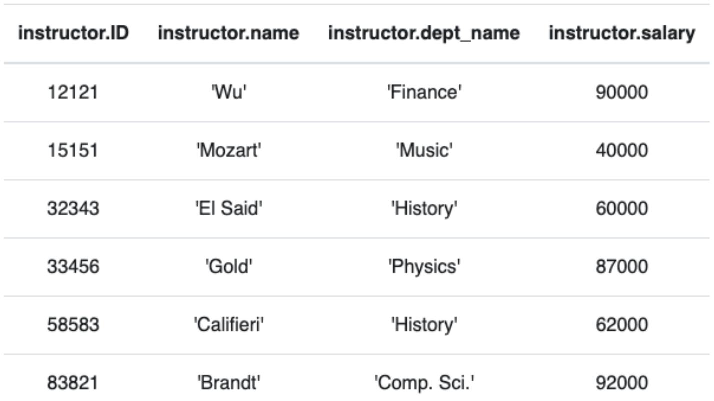
OUTER JOIN

- "An outer join is an operation that combines rows from two or more tables, returning all records from the tables based on a join condition, including unmatched rows filled with `NULL` values. Unlike inner joins that require matches, outer joins preserve data from one or both tables, making them essential for auditing, comparing datasets, and finding missing information."
- There are
  - LEFT OUTER JOIN
  - RIGHT OUTER JOIN
  - FULL OUTER JOIN
- Consider
  - instructor $\bowtie_{ID=i\_id}$ advisor versus
  - instructor ⟕ $_{ID=i\_id}$ advisor
  - Which instructors do not advise any student? $\sigma_{s\_id=null}$ (instructor ⟕ $_{ID=i\_id}$ advisor)

Anti – Join

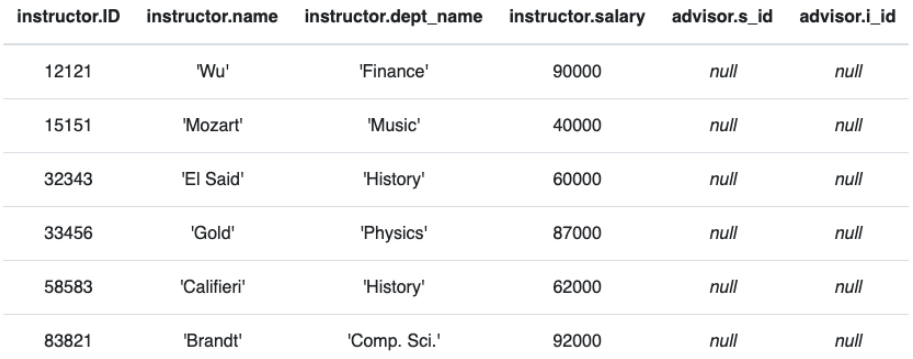
- “An anti-join is when you would like to keep all of the records in the original table except those records that match the other table.”

instructor ▷ ID=i_id advisor



instructor.ID	instructor.name	instructor.dept_name	instructor.salary
12121	'Wu'	'Finance'	90000
15151	'Mozart'	'Music'	40000
32343	'El Said'	'History'	60000
33456	'Gold'	'Physics'	87000
58583	'Califieri'	'History'	62000
83821	'Brandt'	'Comp. Sci.'	92000

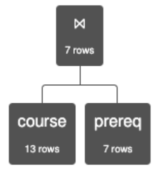
σ i_id=null (instructor ⟕ ID=i_id advisor)



instructor.ID	instructor.name	instructor.dept_name	instructor.salary	advisor.s_id	advisor.i_id
12121	'Wu'	'Finance'	90000	*null*	*null*
15151	'Mozart'	'Music'	40000	*null*	*null*
32343	'El Said'	'History'	60000	*null*	*null*
33456	'Gold'	'Physics'	87000	*null*	*null*
58583	'Califieri'	'History'	62000	*null*	*null*
83821	'Brandt'	'Comp. Sci.'	92000	*null*	*null*

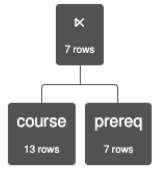
Semi-Join

Semi-join is a type of join that is applied to relations to join them based on the related columns. When semi-join is applied, it returns the rows from one table for which there are matching records in another related table.



course ⋈ prereq

Execution time: 1 ms



course ⋉ prereq

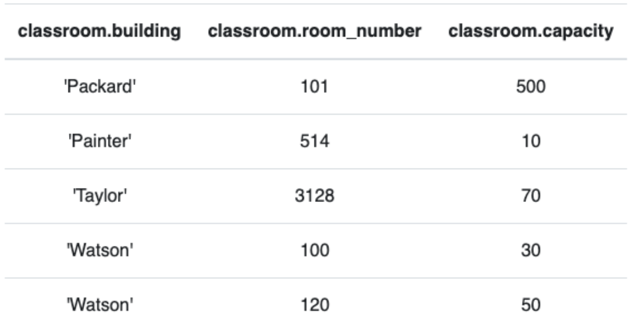
Execution time: 1 ms

course.course_id	course.title	course.dept_name	course.credits	prereq.prereq_id
'BIO-301'	'Genetics'	'Biology'	4	'BIO-101'
'BIO-399'	'Computational Biology'	'Biology'	3	'BIO-101'
'CS-190'	'Game Design'	'Comp. Sci.'	4	'CS-101'
'CS-315'	'Robotics'	'Comp. Sci.'	3	'CS-101'
'CS-319'	'Image Processing'	'Comp. Sci.'	3	'CS-101'
'CS-347'	'Database System Concepts'	'Comp. Sci.'	3	'CS-101'
'EE-181'	'Intro. to Digital Systems'	'Elec. Eng.'	3	'PHY-101'

course.course_id	course.title	course.dept_name	course.credits
'BIO-301'	'Genetics'	'Biology'	4
'BIO-399'	'Computational Biology'	'Biology'	3
'CS-190'	'Game Design'	'Comp. Sci.'	4
'CS-315'	'Robotics'	'Comp. Sci.'	3
'CS-319'	'Image Processing'	'Comp. Sci.'	3
'CS-347'	'Database System Concepts'	'Comp. Sci.'	3
'EE-181'	'Intro. to Digital Systems'	'Elec. Eng.'	3

Group By, Order By

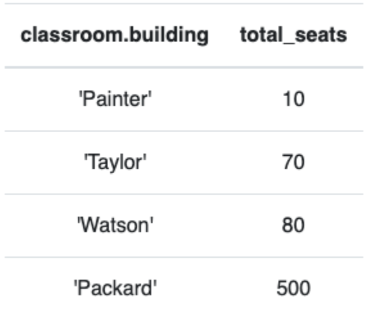
classroom



classroom.building	classroom.room_number	classroom.capacity
'Packard'	101	500
'Painter'	514	10
'Taylor'	3128	70
'Watson'	100	30
'Watson'	120	50

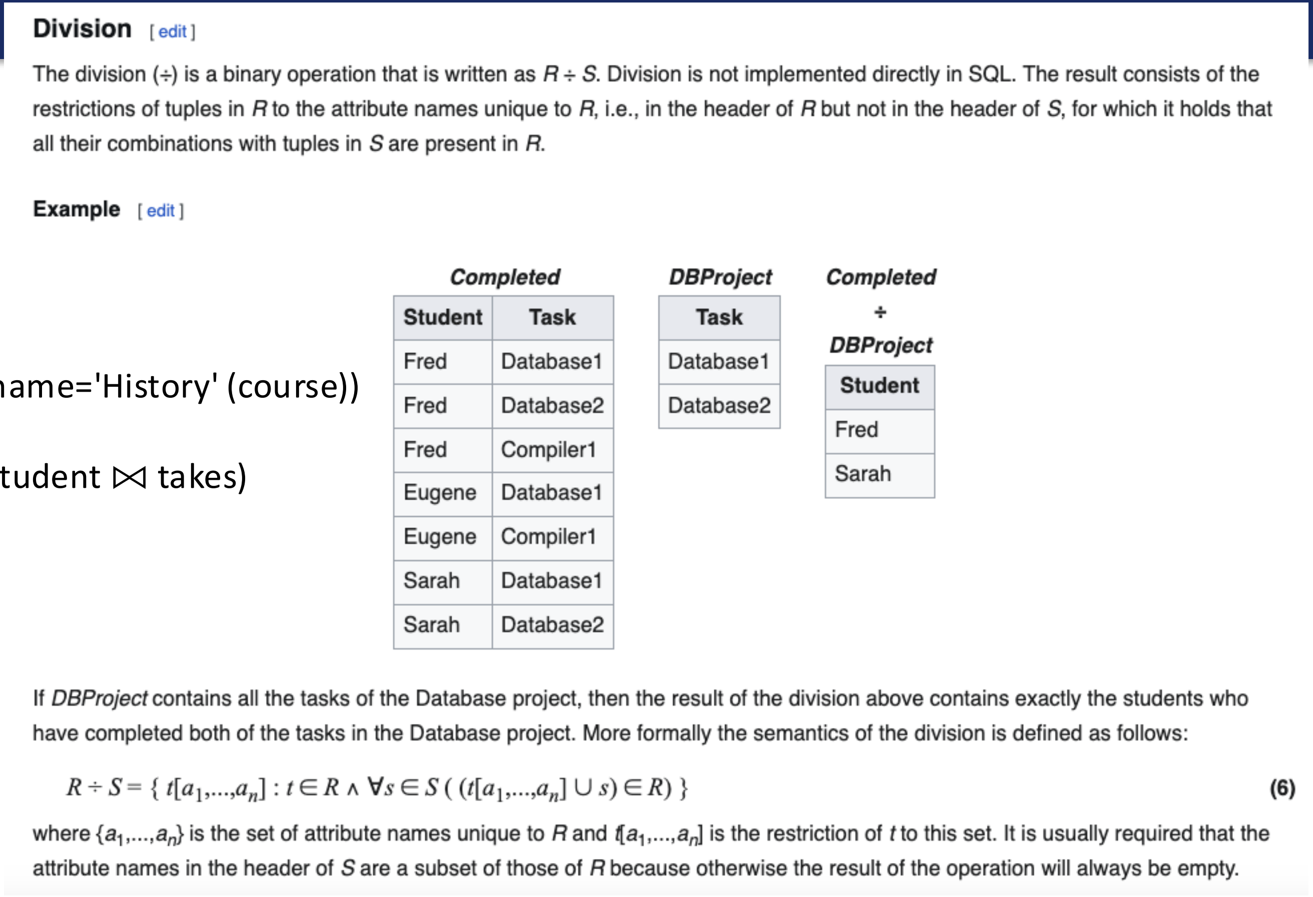
- These are very simple examples.
- We can apply them to relations created by operations on other tables.

τ total_seats (γ building; sum(capacity)→total_seats (classroom))



classroom.building	total_seats
'Painter'	10
'Taylor'	70
'Watson'	80
'Packard'	500

Division



Division [edit]

The division (÷) is a binary operation that is written as $R \div S$. Division is not implemented directly in SQL. The result consists of the restrictions of tuples in *R* to the attribute names unique to *R*, i.e., in the header of *R* but not in the header of *S*, for which it holds that all their combinations with tuples in *S* are present in *R*.

Example [edit]

Completed

Student	Task
Fred	Database1
Fred	Database2
Fred	Compiler1
Eugene	Database1
Eugene	Compiler1
Sarah	Database1
Sarah	Database2

DBProject

Task
Database1
Database2

Completed* ÷ *DBProject

Student
Fred
Sarah

If *DBProject* contains all the tasks of the Database project, then the result of the division above contains exactly the students who have completed both of the tasks in the Database project. More formally the semantics of the division is defined as follows:

$$R \div S = \{ t[a_1,...,a_n] : t \in R \wedge \forall s \in S ((t[a_1,...,a_n] \cup s) \in R) \} \quad (6)$$

where $\{a_1,...,a_n\}$ is the set of attribute names unique to *R* and $t[a_1,...,a_n]$ is the restriction of *t* to this set. It is usually required that the attribute names in the header of *S* are a subset of those of *R* because otherwise the result of the operation will always be empty.

Relax Example

X = π course_id←course_id (σ dept_name='History' (course))

Y = π ID←ID, course_id←course_id (student ⋈ takes)

Y ÷ X

Relational Databases/SQL

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.

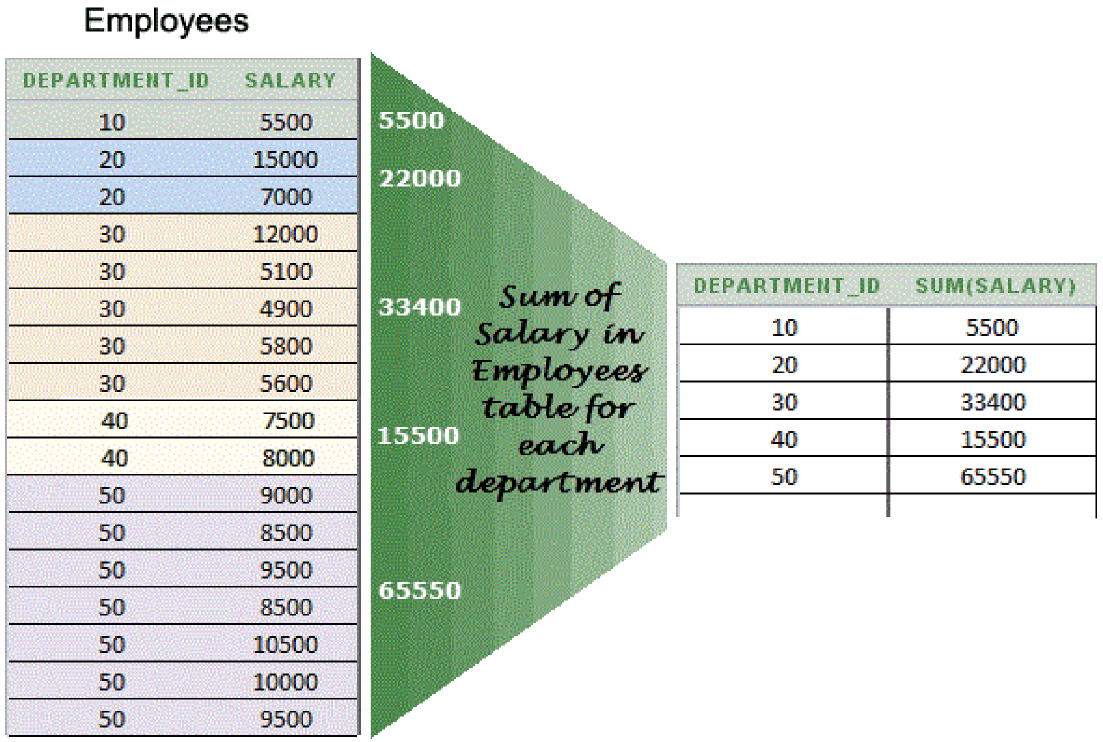
Aggregate Functions – Group By

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

ID	*name*	*dept_name*	*salary*
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

dept_name	*avg_salary*
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

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* ***where*** *dept_name in (‘Biology’, ‘Physics’, ‘Comp.Sci.’)*
  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

- Select …. From …. Where … …
  group by … ….
  Having … …

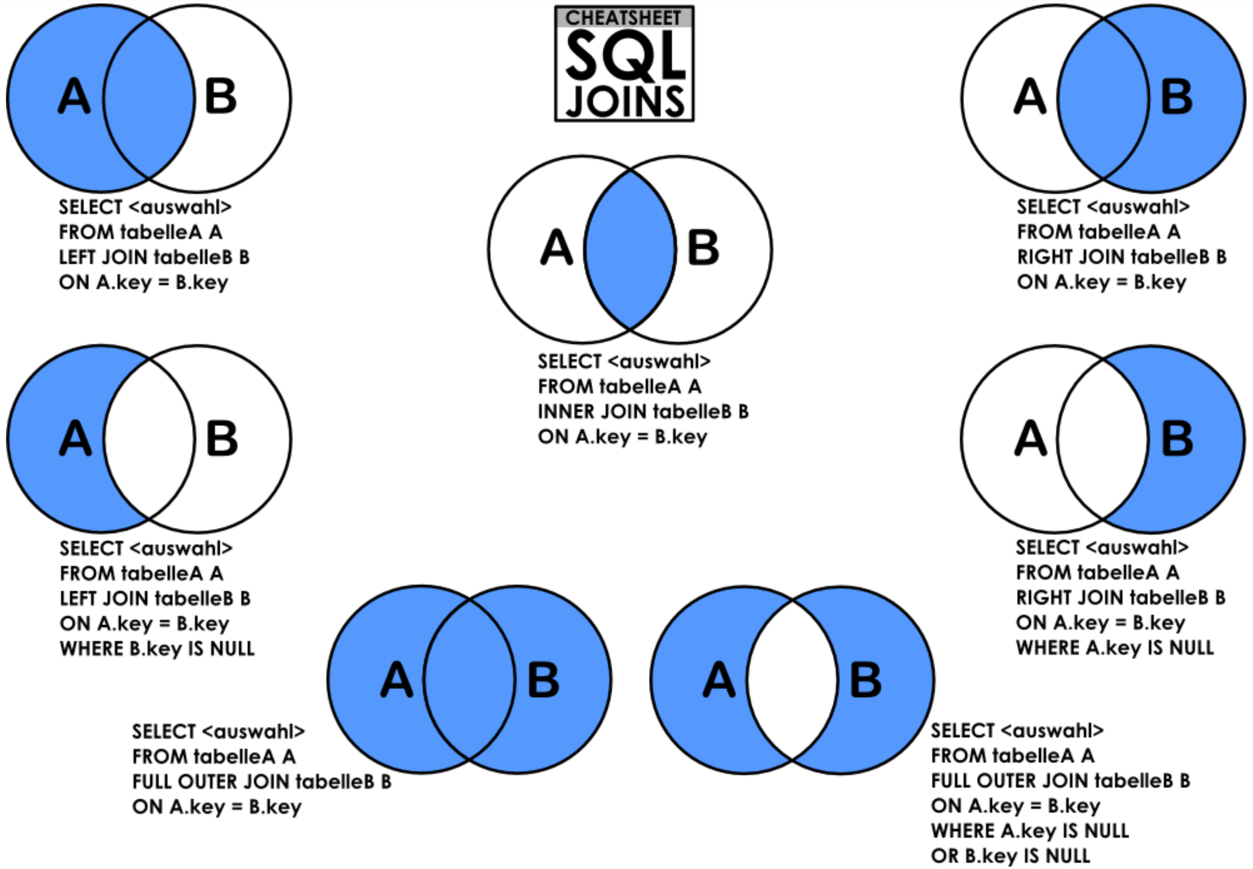
Switch to Notebook

More Function with JOIN

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.
- Three forms of outer join:
  - left outer join
  - right outer join
  - full outer join

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
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
```

Outer Join Examples

- Relation *course*

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

- Relation *prereq*

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

- Observe that

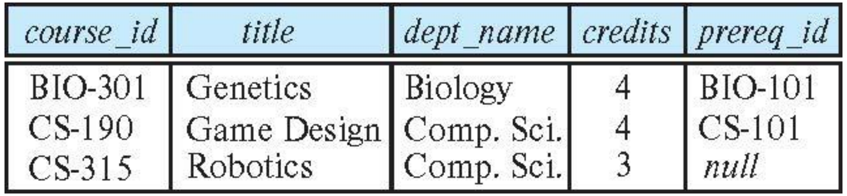
  course information is missing CS-437

  prereq information is missing CS-315

- *x*

Left Outer Join

- *course* **natural left outer join** *prereq*



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

- In relational algebra: *course* ⟕ *prereq*

Right Outer Join

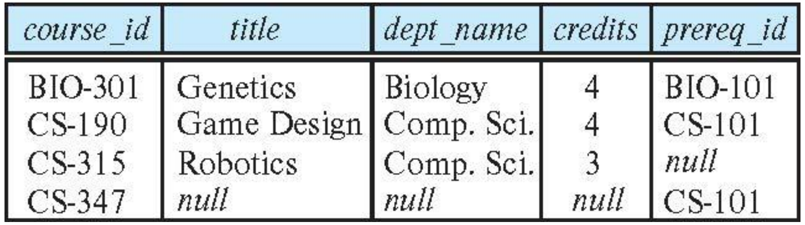
- *course* **natural right outer join** *prereq*

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

- In relational algebra: *course* ⟖ *prereq*

Full Outer Join

- *course* **natural full outer join** *prereq*

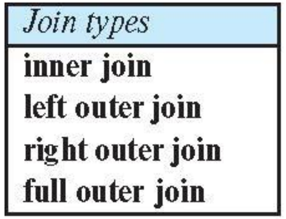


course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

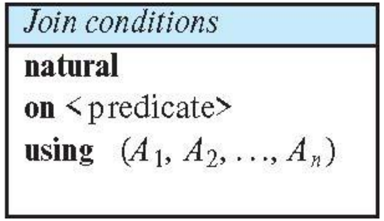
- In relational algebra: *course* ⟗ *prereq*

Joined Types and Conditions

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.



Join types
inner join
left outer join
right outer join
full outer join



Join conditions
natural
on < predicate>
using $(A_1, A_2, \ldots, A_n)$

Joined Relations – Examples

- *course* **natural right outer join** *prereq*

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

- *course* **full outer join** *prereq* **using** (*course_id*)

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

Joined Relations – Examples

- *course* **inner join** *prereq* **on**
  course.course_id = prereq.course_id

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

- What is the difference between the above, and a natural join?
- *course* **left outer join** *prereq* **on**
  course.course_id = prereq.course_id

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

Joined Relations – Examples

- *course* **natural right outer join** *prereq*

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

- *course* **full outer join** *prereq* **using** (*course_id*)

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

Switch to Notebook

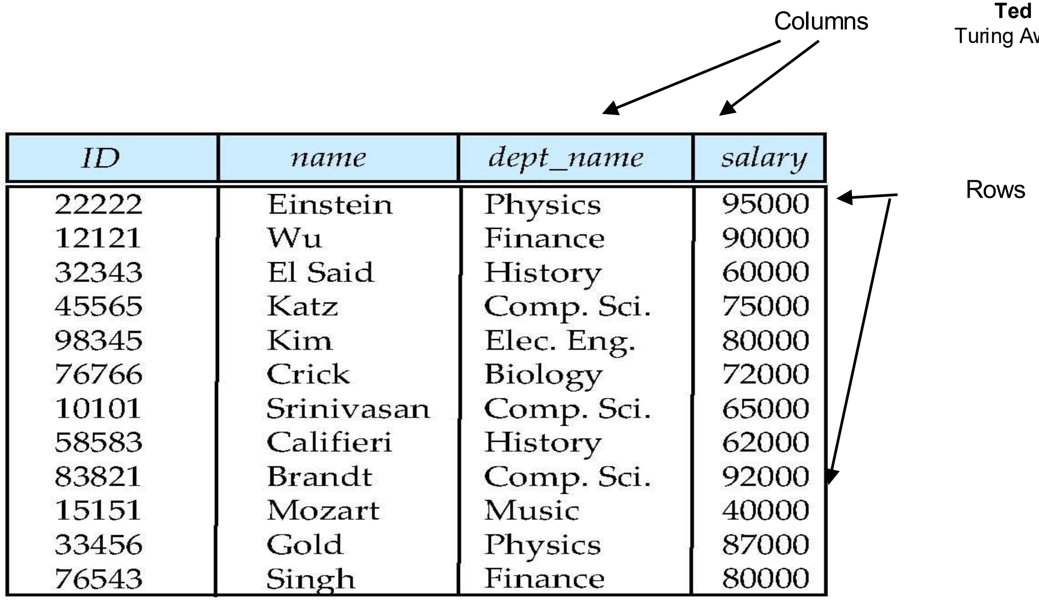
Codd’s 12 Rules

Overview

Relational Model

- All the data is stored in various tables.
- Example of tabular data in the relational model

Ted Codd
Turing Award 1981


Columns
Rows

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

(a) The *instructor* table

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.

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 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 1 – Information Rule

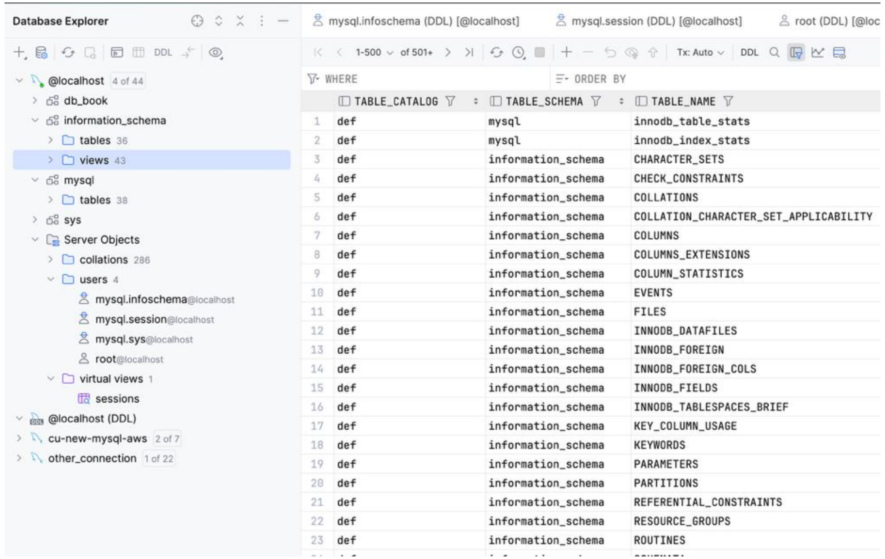
- Rule 1: Information Rule – The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.
- We have seen that all user application data is stored in tables.
  We will see metadata in a minute through the *online catalog.*
- What other data is there? Every database implementation handles this differently.
- MySQL has 3 schemas/databases that you have seen but not examine.
  - *information_schema* (is the catalog, which we will discuss in a minute).
  - *sys*
  - *mysql*

  That contain a mix of metadata, security information, configuration and performance data, … …
- *Show the schema in DataGrip.*

Some Interesting Queries

- sys
  - select * from sys.sys_config;
  - select * from sys.host_summary;
  - select * from sys.session;
  - select * from sys.user_summary;
- mysql
  - select * from mysql.help_topic;
  - select * from mysql.user;
  - select * from mysql.default_roles;
  - select * from mysql.innodb_table_stats;
- Most of this is pretty obscure, and all empty/incomplete in a single server, free version of MySQL.

Metadata and System Tables



Database Explorer

mysql.infoschema (DDL) [@localhost] | mysql.session (DDL) [@localhost] | root (DDL) [@loc

1-500 of 501+ | Tx: Auto | DDL

WHERE | ORDER BY

- @localhost 4 of 44
  - db_book
  - information_schema
    - tables 36
    - views 43
  - mysql
    - tables 38
  - sys
  - Server Objects
    - collations 286
    - users 4
      - mysql.infoschema@localhost
      - mysql.session@localhost
      - mysql.sys@localhost
      - root@localhost
    - virtual views 1
      - sessions
- @localhost (DDL)
- cu-new-mysql-aws 2 of 7
- other_connection 1 of 22

	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME
1	def	mysql	innodb_table_stats
2	def	mysql	innodb_index_stats
3	def	information_schema	CHARACTER_SETS
4	def	information_schema	CHECK_CONSTRAINTS
5	def	information_schema	COLLATIONS
6	def	information_schema	COLLATION_CHARACTER_SET_APPLICABILITY
7	def	information_schema	COLUMNS
8	def	information_schema	COLUMNS_EXTENSIONS
9	def	information_schema	COLUMN_STATISTICS
10	def	information_schema	EVENTS
11	def	information_schema	FILES
12	def	information_schema	INNODB_DATAFILES
13	def	information_schema	INNODB_FOREIGN
14	def	information_schema	INNODB_FOREIGN_COLS
15	def	information_schema	INNODB_FIELDS
16	def	information_schema	INNODB_TABLESPACES_BRIEF
17	def	information_schema	KEY_COLUMN_USAGE
18	def	information_schema	KEYWORDS
19	def	information_schema	PARAMETERS
20	def	information_schema	PARTITIONS
21	def	information_schema	REFERENTIAL_CONSTRAINTS
22	def	information_schema	RESOURCE_GROUPS
23	def	information_schema	ROUTINES

- db_book is a schema that holds “data.”
- information_schema is the “online catalog” and holds information about schemas, tables, columns, keys, indexes, ……
- mysql: “The mysql schema is the system schema. It contains tables that store information required by the MySQL server as it runs. … (https://dev.mysql.com/doc/refman/8.4/en/system-schema.html)
- sys: “MySQL 8.4 includes the sys schema, a set of objects that helps DBAs and developers interpret data collected by the Performance Schema.” (https://dev.mysql.com/doc/refman/8.4/en/sys-schema.html)
- Server Objects contains information about sessions, collations, … …

Data Definition Language (DDL)

- Specification notation for defining the database schema

  Example: **create table** *instructor* (
  ID **char**(5),
  name **varchar**(20),
  dept_name **varchar**(20),
  salary **numeric**(8,2))

- DDL compiler generates a set of table templates stored in a ***data dictionary***
- Data dictionary contains metadata (i.e., data about data)
  - Database schema
  - Integrity constraints
    - Primary key (ID uniquely identifies instructors)
  - Authorization
    - Who can access what

Metadata and Catalog

- ‘Metadata is "data that provides information about other data". In other words, it is "data about data". Many distinct types of metadata exist, including descriptive metadata, structural metadata, administrative metadata, reference metadata and statistical metadata.’ (https://en.wikipedia.org/wiki/Metadata)
- “The database catalog of a database instance consists of metadata in which definitions of database objects such as base tables, views (virtual tables), synonyms, value ranges, indexes, users, and user groups are stored. ... ...

  The SQL standard specifies a uniform means to access the catalog, called the INFORMATION_SCHEMA, but not all databases follow this ...” (https://en.wikipedia.org/wiki/Database_catalog)
- Codd’s Rule 4: Dynamic online catalog based on the relational model:
  - The data base description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.

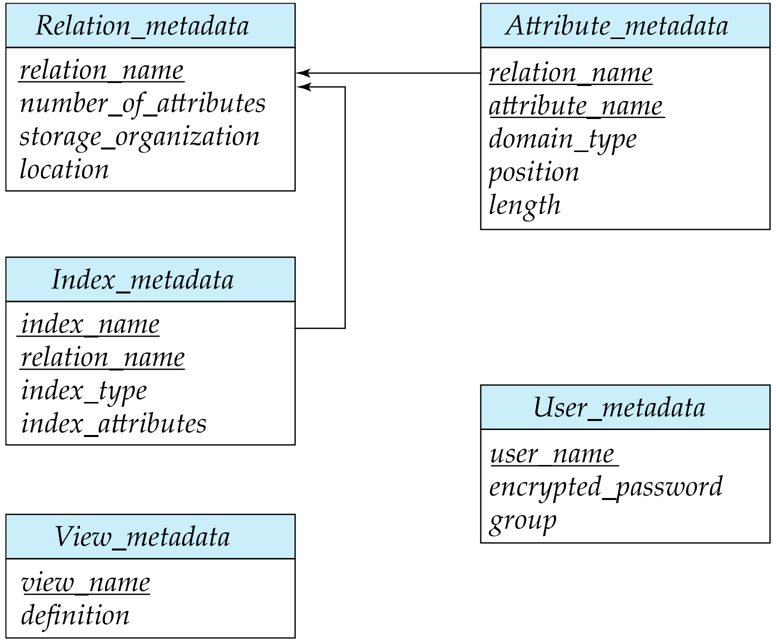
Data Dictionary Storage

The **Data dictionary** (also called **system catalog**) stores **metadata**; that is, data about data, such as

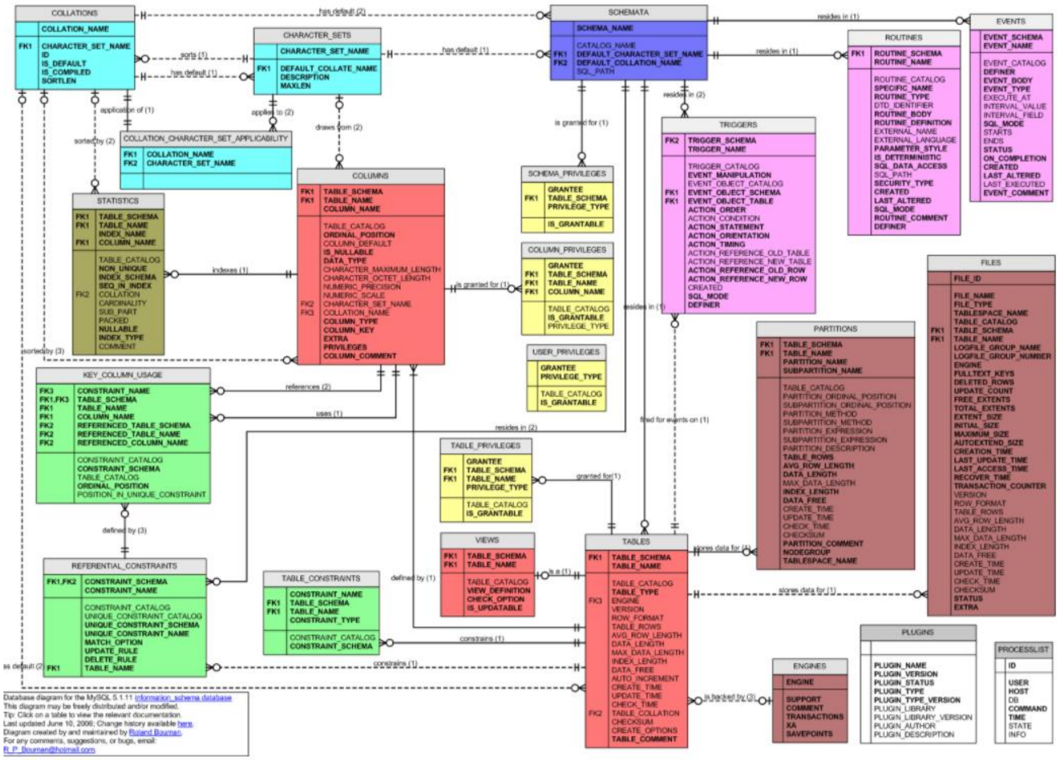
- Information about relations
  - names of relations
  - names, types and lengths of attributes of each relation
  - names and definitions of views
  - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
  - number of tuples in each relation
- Physical file organization information
  - How relation is stored (sequential/hash/…)
  - Physical location of relation
- Information about indices (Chapter 14)

Relational Representation of System Metadata

- Relational representation on disk
- Specialized data structures designed for efficient access, in memory



MySQL Catalog (Information_Schema)



Some of the MySQL Information Schema Tables:

- 'ADMINISTRABLE_ROLE_AUTHORIZATIONS'
- 'APPLICABLE_ROLES'
- 'CHARACTER_SETS'
- 'CHECK_CONSTRAINTS'
- 'COLUMN_PRIVILEGES'
- 'COLUMN_STATISTICS'
- 'COLUMNS'
- 'ENABLED_ROLES'
- 'ENGINES'
- 'EVENTS'
- 'FILES'
- 'KEY_COLUMN_USAGE'
- 'PARAMETERS'
- 'REFERENTIAL_CONSTRAINTS'
- 'RESOURCE_GROUPS'
- 'ROLE_COLUMN_GRANTS'
- 'ROLE_ROUTINE_GRANTS'
- 'ROLE_TABLE_GRANTS'
- 'ROUTINES'
- 'SCHEMA_PRIVILEGES'
- 'STATISTICS'
- 'TABLE_CONSTRAINTS'
- 'TABLE_PRIVILEGES'
- 'TABLES'
- 'TABLESPACES'
- 'TRIGGERS'
- 'USER_PRIVILEGES'
- 'VIEW_ROUTINE_USAGE'
- 'VIEW_TABLE_USAGE'
- 'VIEWS'

- CREATE and ALTER statements modify the data.
- DBMS reads information:
  - Parsing
  - Optimizer
  - etc.

Switch to Notebook

Rule 6: View Updating Rule

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

Which can theoretically be updated

What are some examples of things that make a view no “updatable?”

- Aggregate Functions
  - SQL INSERT, UPDATE and DELETE affect one row at a time.
  - Aggregate functions convert multiple rows into one row, which may invove sum, average, … on column values.
  - How would you map the update of an average to the individual, underlying rows?
- The underlying table may have NOT NULL columns that are not in the view ➔ INSERT will not work.
- The “project” can merge or transform columns in rows, e.g.
  - CREATE VIEW cool as
    SELECT concat(first_name, last_name) as full_name from person.
  - What would it mean to do an UPDATE on cool.full_name

Overview

Index Creation

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to read every record to find a record with particular value
- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.
- We create an index with the **create index** command

  create index <name> **on** <relation-name> (attribute);

Index Creation Example

- **create table** *student*
  (*ID* **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))
- **create index** *studentID_index* **on** *student*(*ID*)
- The query:

  select *
  from *student*
  where *ID* = '12345'

  can be executed by using the index to find the required record, without looking at all records of *student*

DFF:

- An index on (column1, column2, column3)
- Is also an index on (column1) and (column1, column2)
- But not (column2), (column3), (column2, column3).
- We will see "why" later in the semester.

Show performance in Notebook

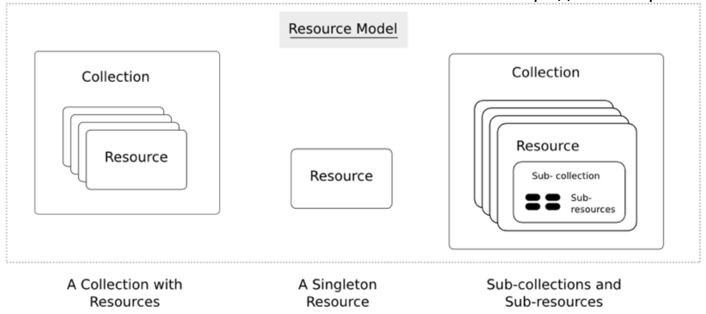
Applications, REST, Data Engineering Project

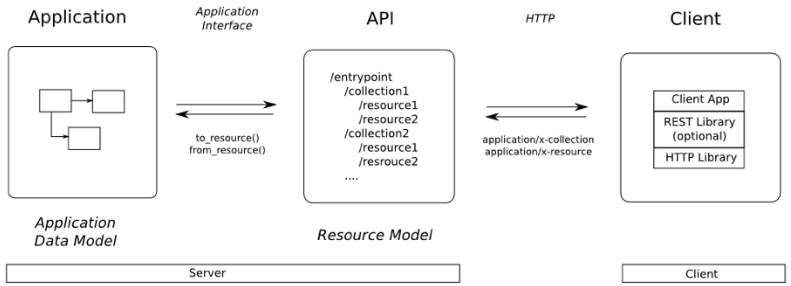
REST and Web Applications

Web Application Problem Statement

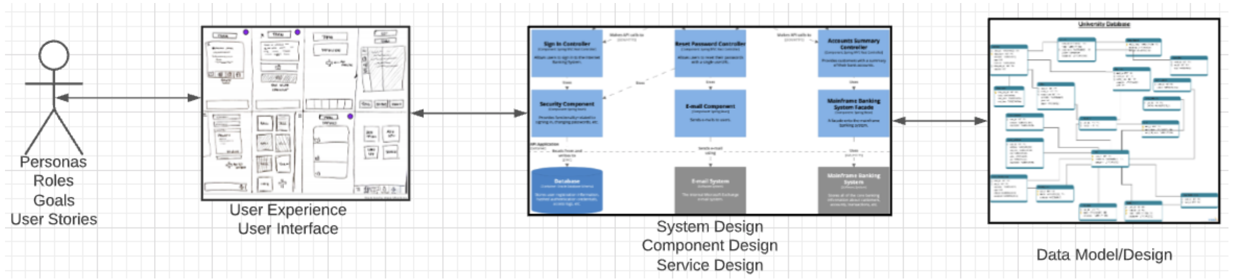
- We must build a system that supports create, retrieve, update and delete for IMDB and Game of Thrones Datasets.
- This requires implementing *create, retrieve, update and delete (CRUD)* for resources.

https://restful-api-design.readthedocs.io/en/latest/resources.html





- We will design, develop, test and deploy the system iteratively and continuously.
- There are four core domains.

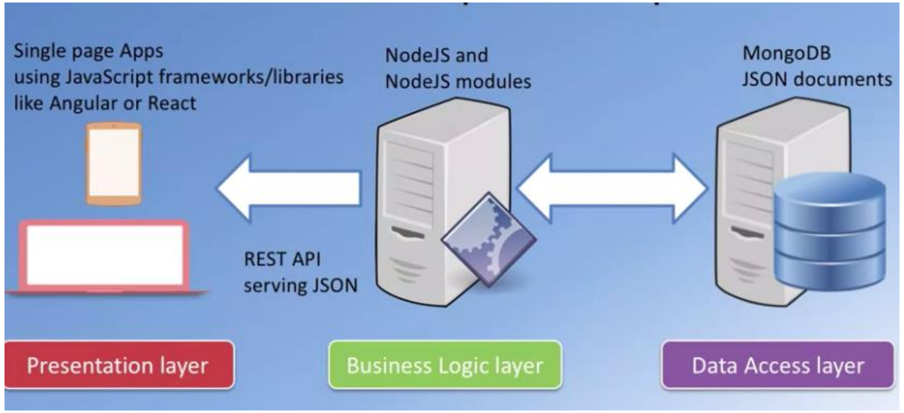


- In this course,
- We focus on the data dimension.
- We will get some insight into the other dimensions.

Interactive/Operational

- "A full stack web developer is a person who can develop both client and server software. In addition to mastering HTML and CSS, he/she also knows how to:
  - Program a browser (like using JavaScript, jQuery, Angular, or Vue)
  - Program a server (like using PHP, ASP, Python, or Node)
  - Program a database (like using SQL, SQLite, or MongoDB)"

  https://www.w3schools.com/whatis/whatis_fullstack.asp
- We will do a simple full stack app.
  - Three databases:
    - MySQL
    - MongoDB
    - Neo4j
  - The application tier will be Python and FastAPI.
  - The web UI will be Angular.
  - The primary focus is the data layer and application layer that access it.
  - I will provide a simple UI and template.

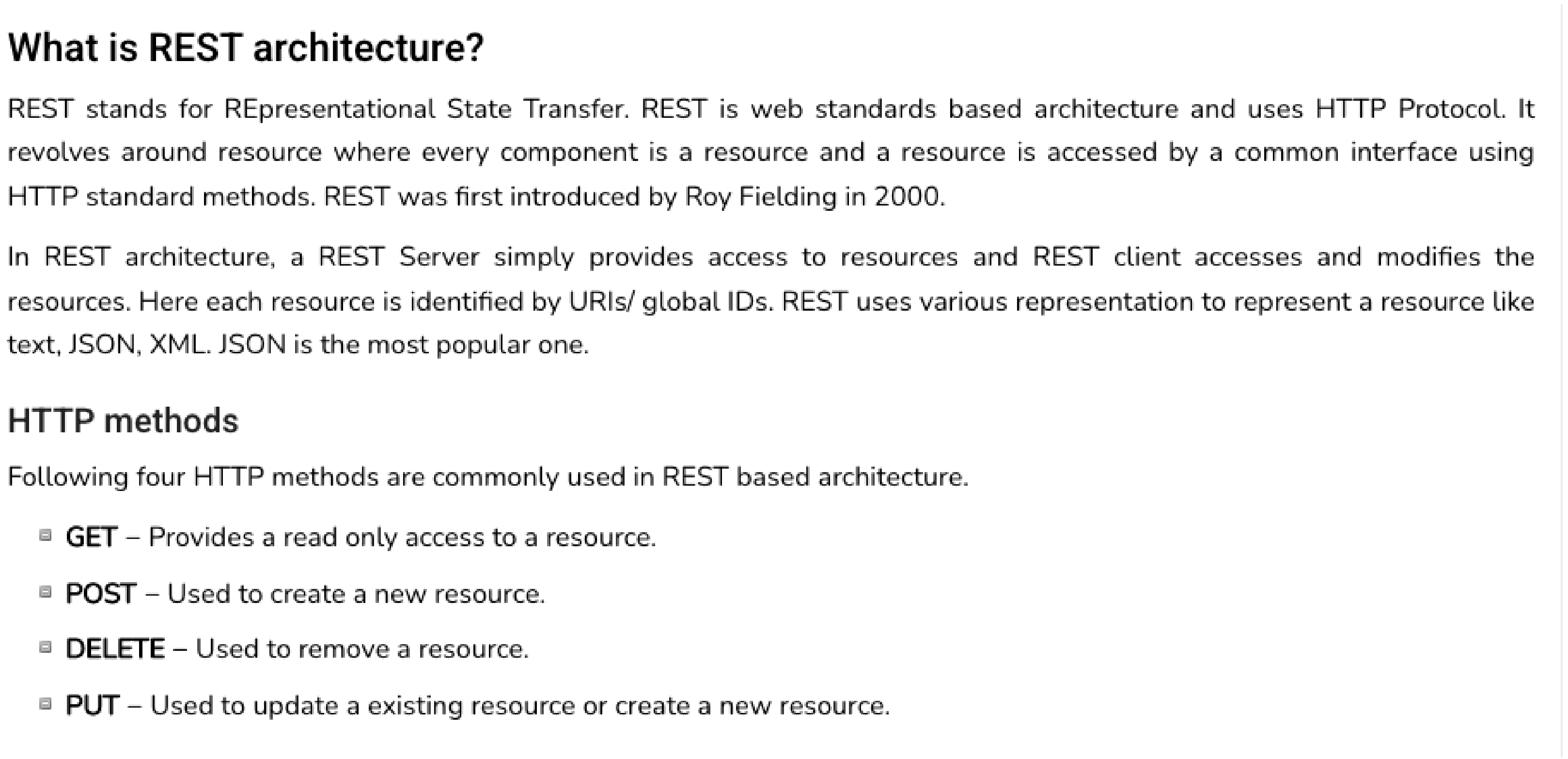

Single page Apps using JavaScript frameworks/libraries like Angular or React
NodeJS and NodeJS modules
MongoDB JSON documents
REST API serving JSON
Presentation layer
Business Logic layer
Data Access layer

Data Modeling Concepts and REST

Almost any data model has the same core concepts:

- Types and instances:
  - Entity Type: A definition of a type of thing with properties and relationships.
  - Entity Instance: A specific instantiation of the Entity Type
  - Entity Set Instance: An Entity Type that:
    - Has properties and relationships like any entity, but …
    - Has at least one *special relationship* – ***contains.***
- Operations, minimally CRUD, that manipulate entity types and instances:
  - Create
  - Retrieve
  - Update
  - Delete
  - Reference/Identify/… …
  - Host/database/table/pk

REST (https://www.tutorialspoint.com/restful/restful_introduction.htm)



What is REST architecture?

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

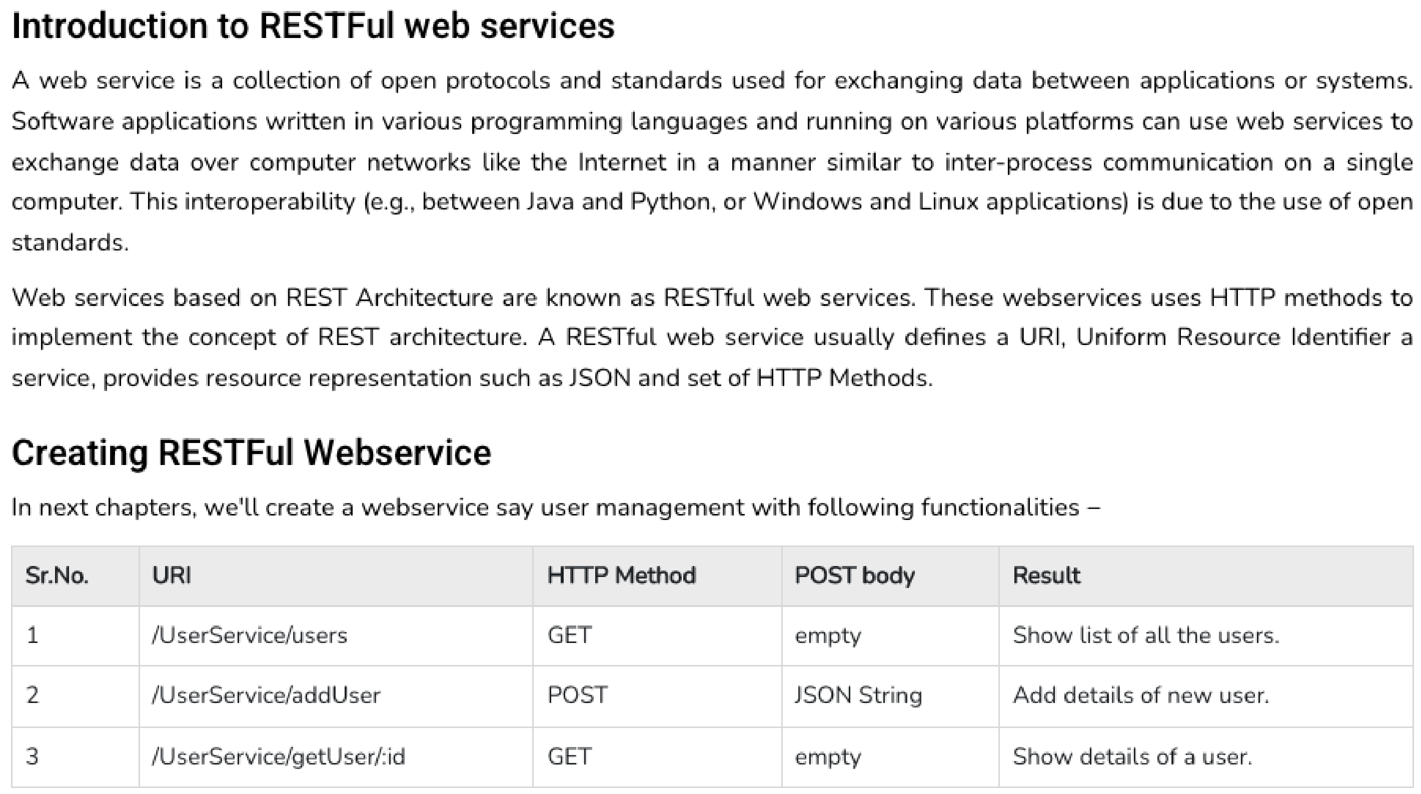
In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

- **GET** – Provides a read only access to a resource.
- **POST** – Used to create a new resource.
- **DELETE** – Used to remove a resource.
- **PUT** – Used to update a existing resource or create a new resource.

REST (https://www.tutorialspoint.com/restful/restful_introduction.htm)



Introduction to RESTFul web services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

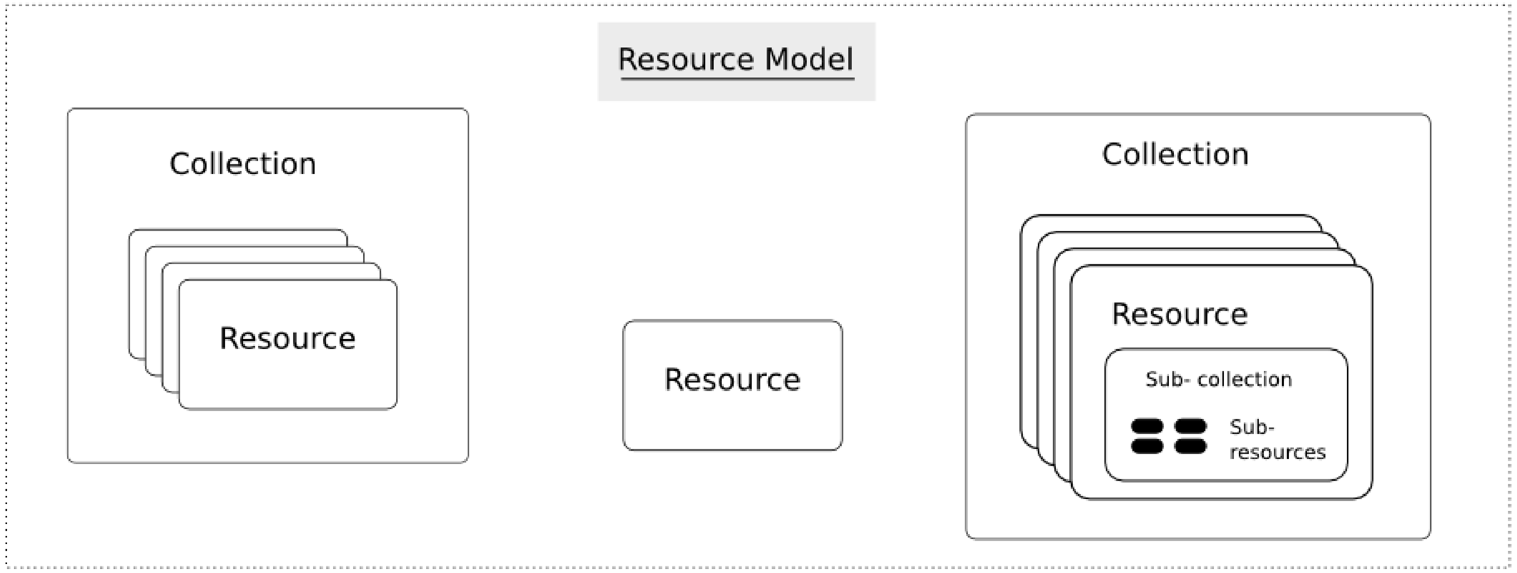
Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP Methods.

Creating RESTFul Webservice

In next chapters, we'll create a webservice say user management with following functionalities –

Sr.No.	URI	HTTP Method	POST body	Result
1	/UserService/users	GET	empty	Show list of all the users.
2	/UserService/addUser	POST	JSON String	Add details of new user.
3	/UserService/getUser/:id	GET	empty	Show details of a user.

REST and Resources

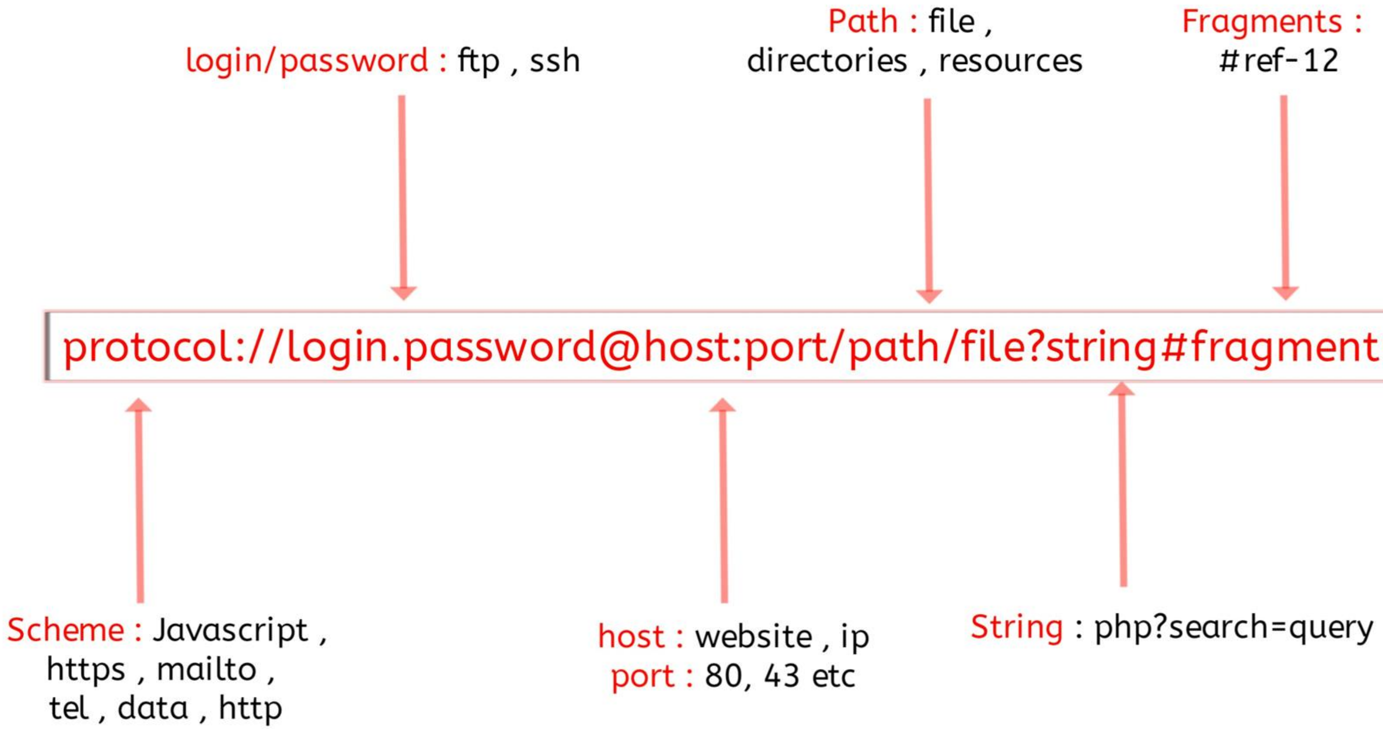


A Collection with Resources

A Singleton Resource

Sub-collections and Sub-resources

URLs


login/password : ftp , ssh
Path : file , directories , resources
Fragments : #ref-12
protocol://login.password@host:port/path/file?string#fragment
Scheme : Javascript , https , mailto , tel , data , http
host : website , ip
port : 80, 43 etc
String : php?search=query

Code

- /Users/donald.ferguson/Dropbox/000/000-A-Current-Examples/W4111-FastAPI-IMDB-Solution
- /Users/donald.ferguson/Dropbox/000/000-A-Current-Examples/current-dashboard