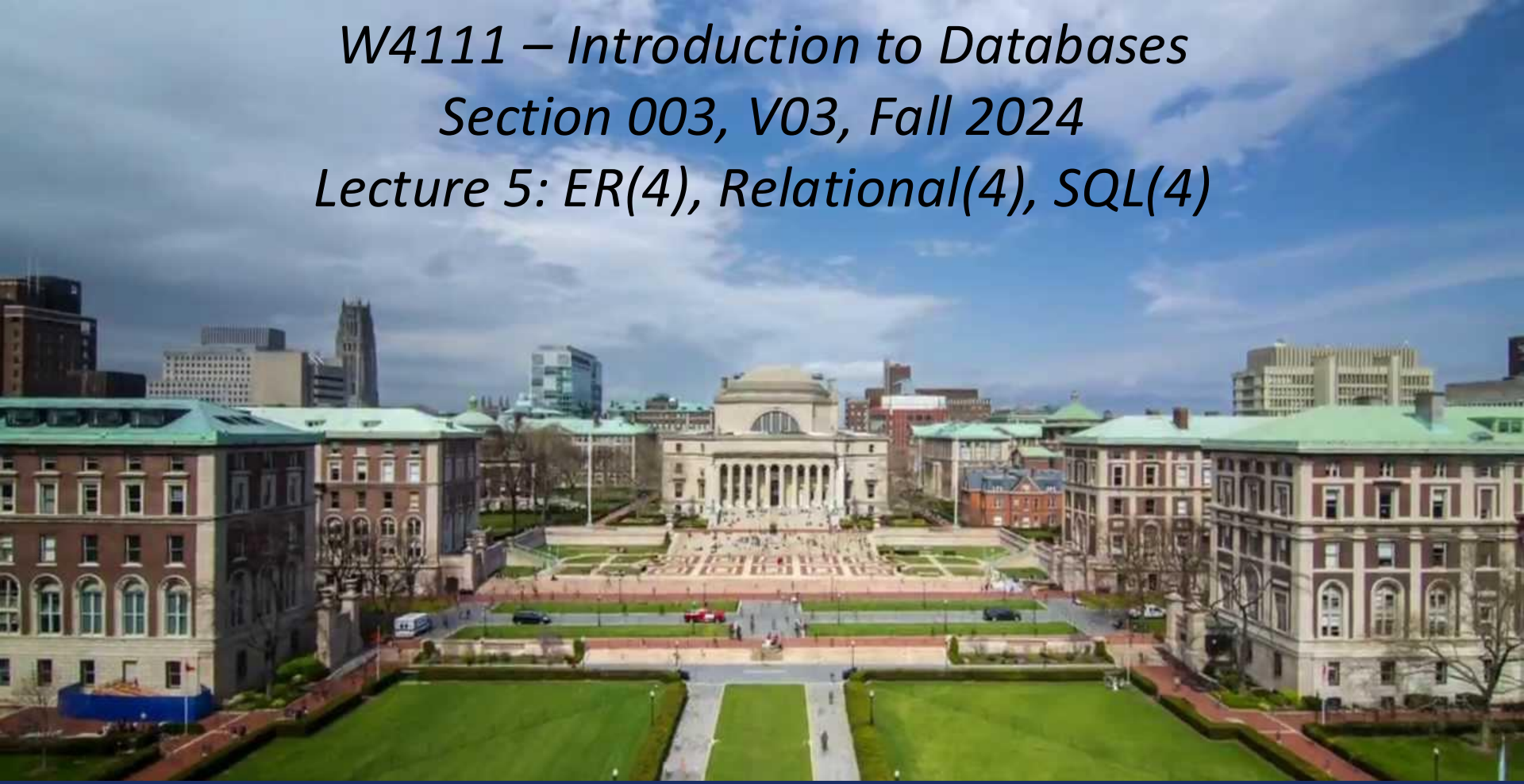
W4111 – Introduction to Databases
Section 003, V03, Fall 2024
Lecture 5: ER(4), Relational(4), SQL(4)



W4111 – Introduction to Databases
Section 003, V03, Fall 2024
Lecture 5: ER(4), Relational(4), SQL(4)

We will start in a few minutes.

Today's Contents

Contents

- Introduction and course updates.
- ER Modeling (4).
- Relational model and algebra (4).
- SQL (4).
- Homework and projects.

Introduction and course updates

Course Updates

- Questions?
- Homework 2
  - Will come out in two parts.
  - HW2a this weekend and due in one week.
  - HW2b midweek and due after the midterm.
    - Working on HW2b is excellent study prep for the practical questions on the midterm.
    - Part of HW2b will also be the nucleus of your “project.”
- Midterm Exam
  - 18-OCT in the classroom from 10:10 to 11:30.
  - Written exam “on paper.”
  - No electronics.
  - We will document what paper, books, etc. you can use for “open book.”

ER Modeling

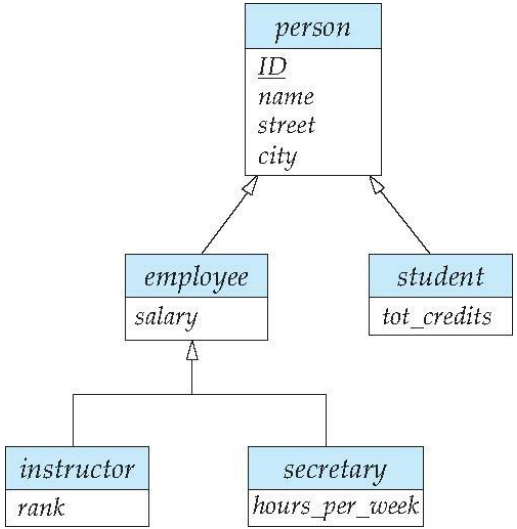
Inheritance

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.

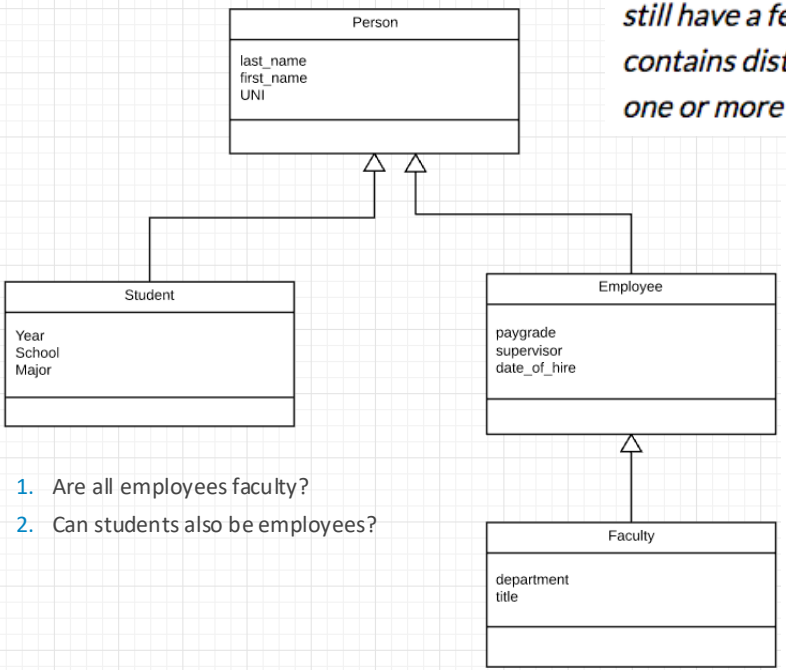
Specialization Example

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



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.

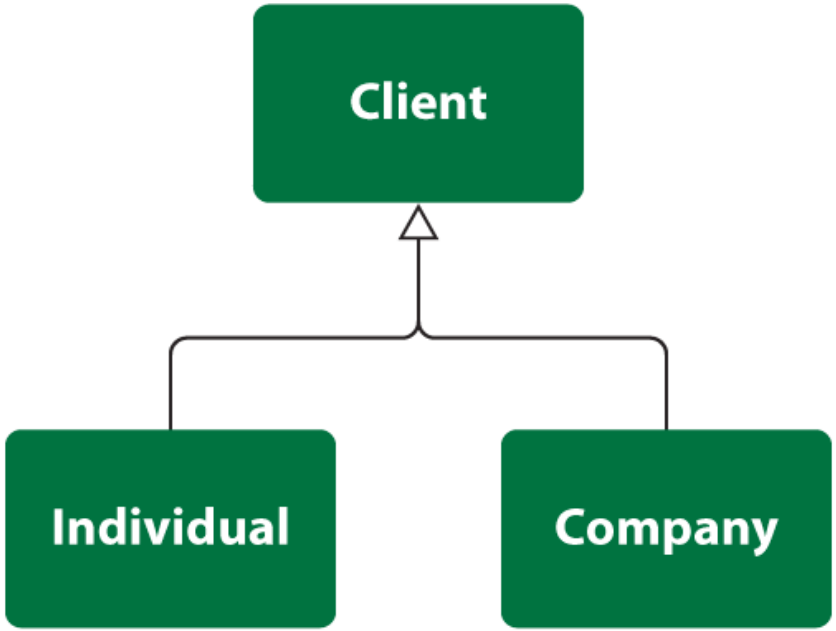
2 disjoint/overlapping

- 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).

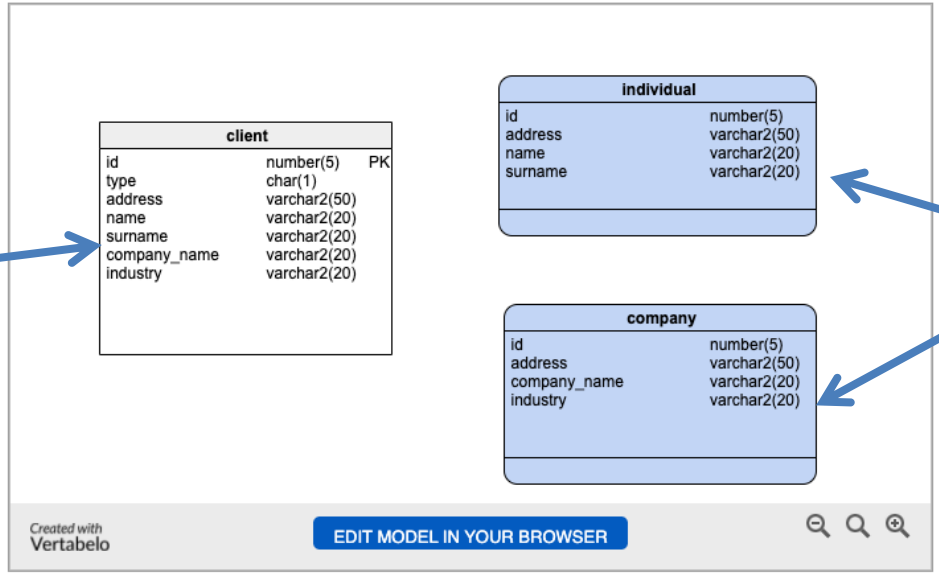


One Table

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`:



Tables

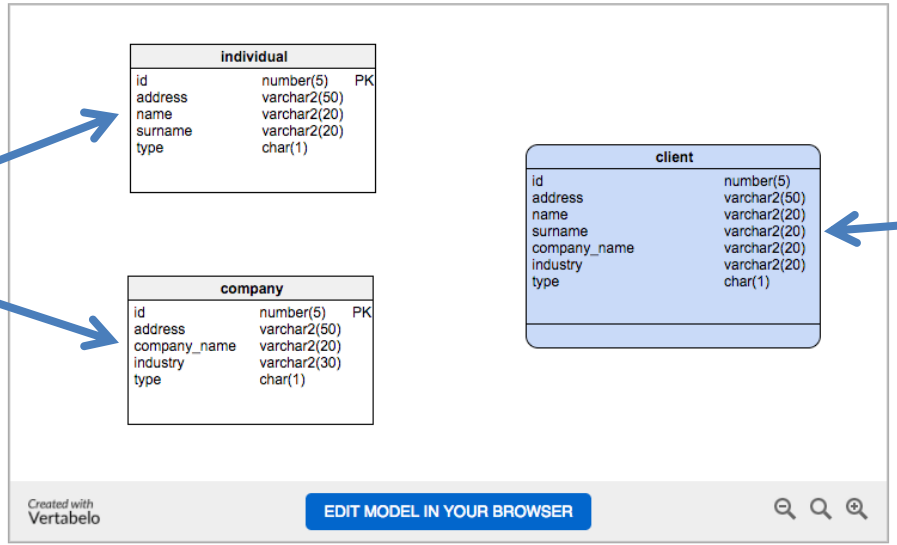
Views

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.

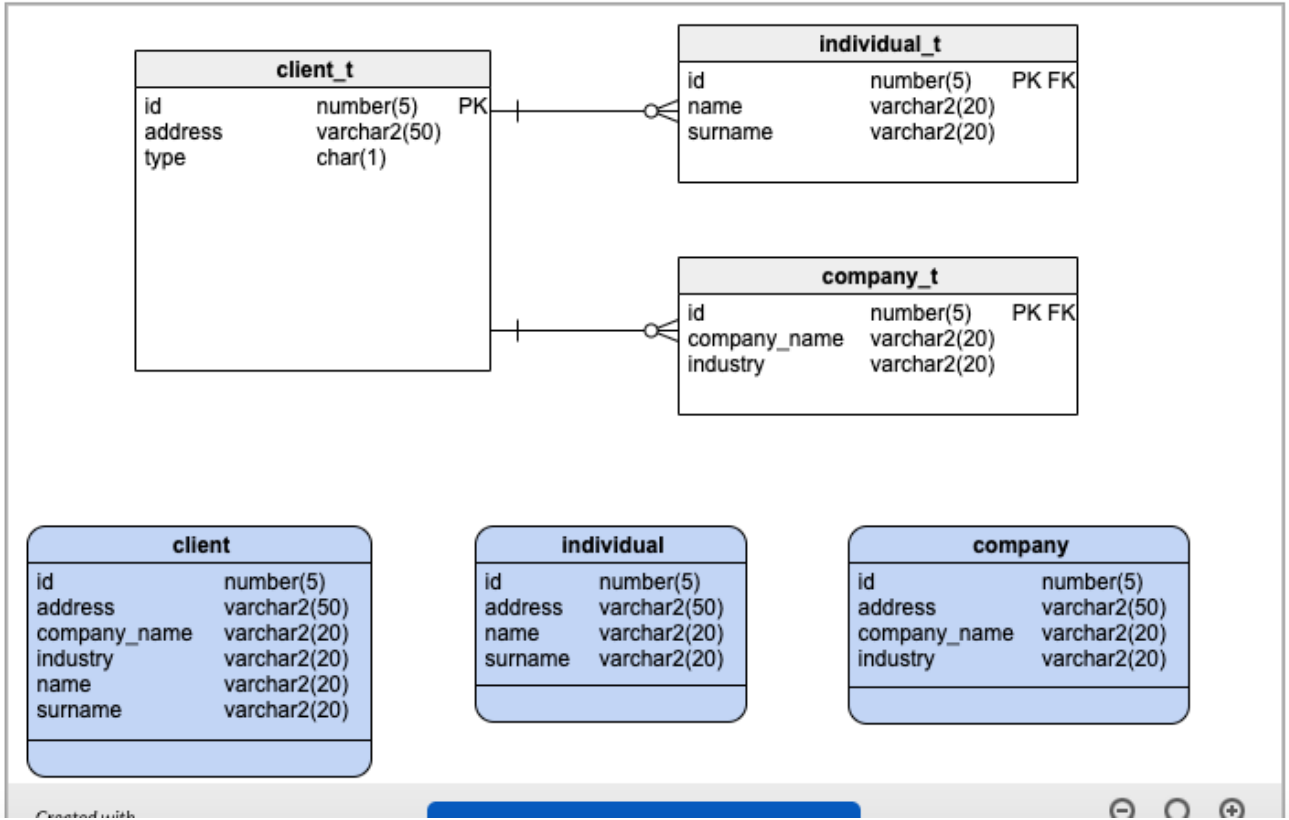
The diagram below presents two tables, `individual` and `company`, and a view (the blue one) called `client`.



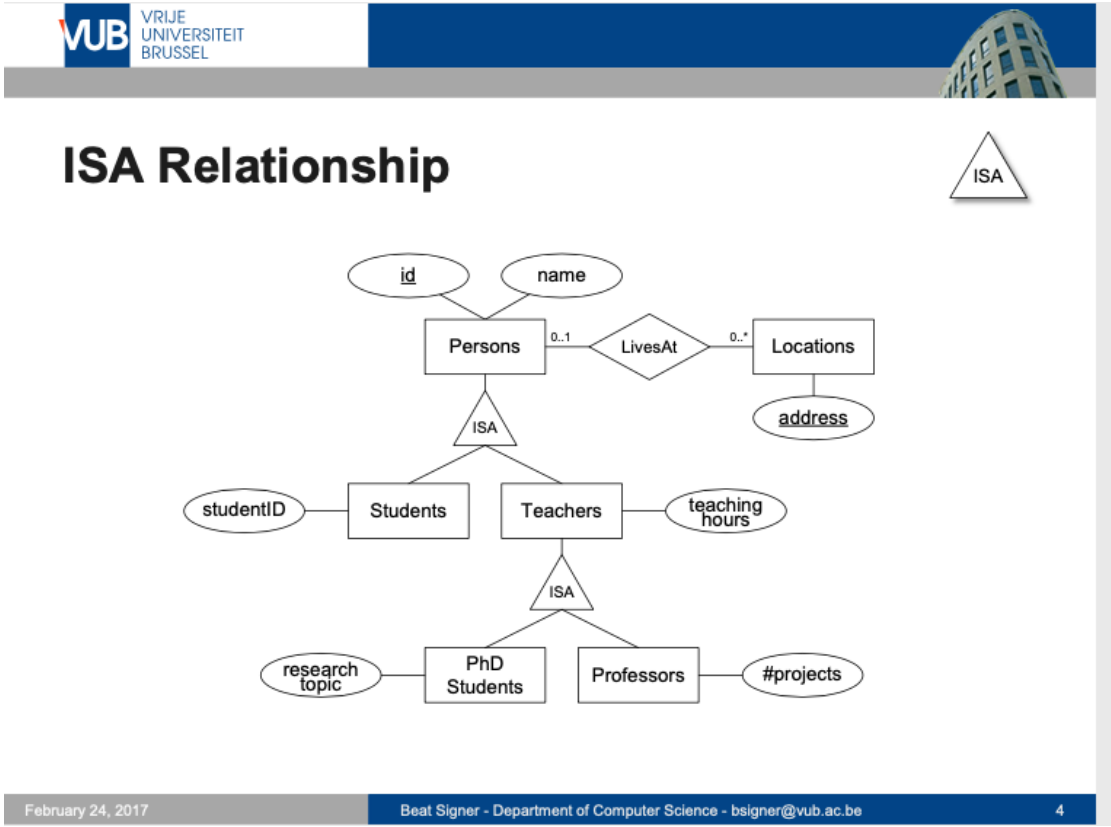
Tables

View

Three Table

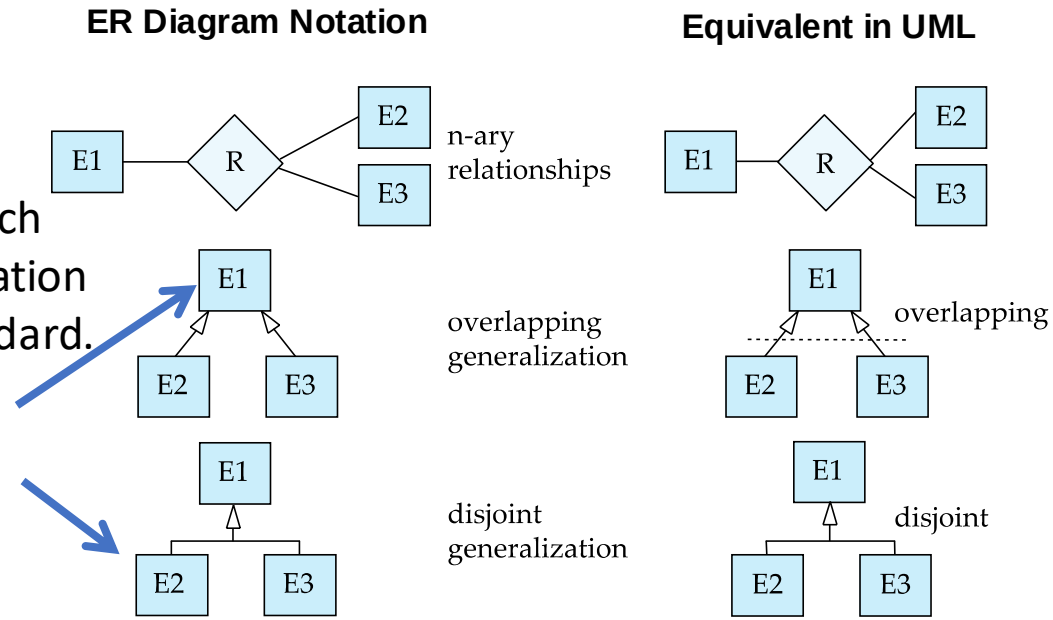


ISA Relationship



ER vs. UML Class Diagrams

I use this approach in Crow's Foot Notation but that is not standard.



* Generalization can use merged or separate arrows independent of disjoint/overlapping

Complex Attributes

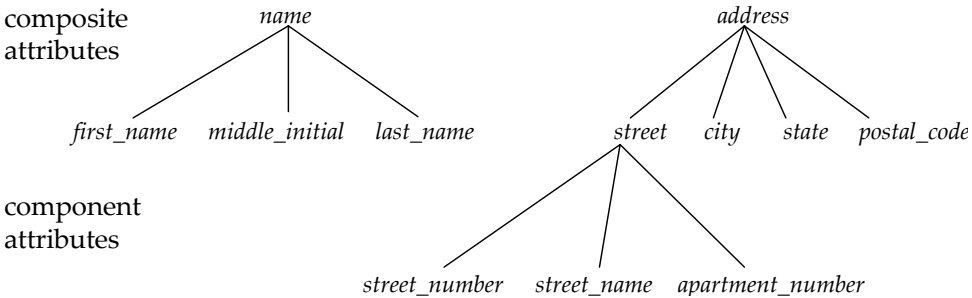
Complex Attributes

- 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

Not covered

Composite Attributes

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



Not covered

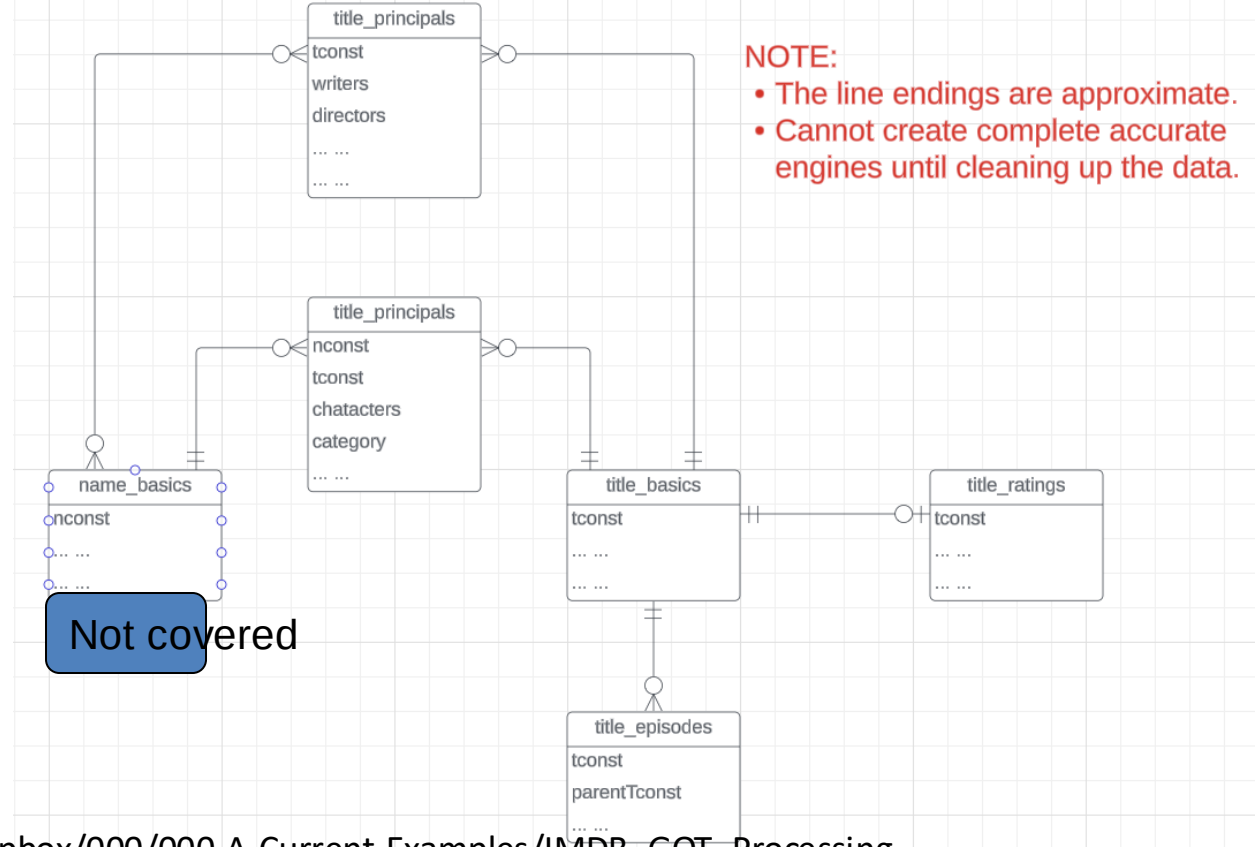
Entity Attributes

- The relational model and well-designed SQL schema have *atomic attributes.*
- There are several ways to think about attributes:
  - Simple (Atomic) vs Composite
  - Single Valued versus Multi-valued
  - Derived or Not Derived
- And, there can be combinations, for example a Composite, Multi-Value Attribute. Phone number is an example:
  - A phone number is a composite (+1, 914-555-1212)
  - A customer may have several: work, home, mobile, … …
- Examining *name_basics* from the IMDB dataset is interesting.

Not covered

IMDB Free Data

- IMDB Free Dataset is a set of TSV files:
  - Title Basics
  - Title AKAS
  - Title Crew
  - Title Episodes
  - Title Principals
  - Title Ratings
  - Name Basics
- The data needs a lot of “clean up.”



See /Users/donald.ferguson/Dropbox/000/000-A-Current-Examples/IMDB_GOT_Processing

Example from IMDB

Switch to notebook

- Consider name_basics

10 rows | Tx: Auto | DDL | CSV

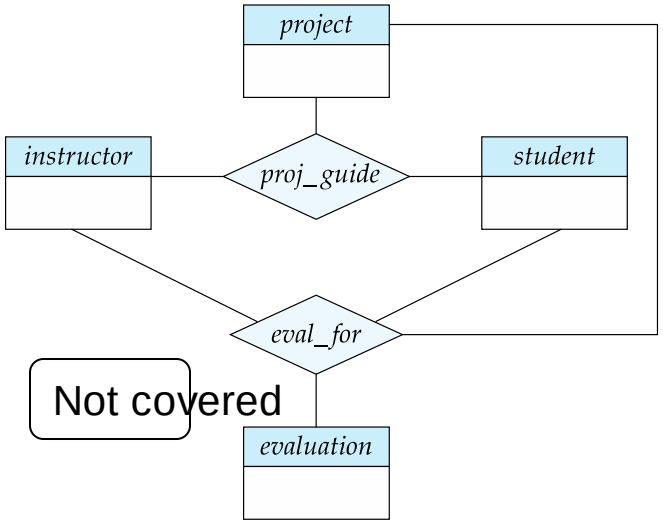
	nconst	primaryName	birthYear	deathYear	primaryProfession	knownForTitles
1	nm0000001	Fred Astaire	1899	1987	soundtrack,actor,miscellaneous	tt0050419,tt0031983,tt0072308,tt0053137
2	nm0000002	Lauren Bacall	1924	2014	actress,soundtrack	tt0071877,tt0117057,tt0037382,tt0038355
3	nm0000003	Brigitte Bardot	1934	<null>	actress,soundtrack,music_department	tt0049189,tt0056404,tt0057345,tt0054452
4	nm0000004	John Belushi	1949	1982	actor,soundtrack,writer	tt0077975,tt0072562,tt0080455,tt0078723
5	nm0000005	Ingmar Bergman	1918	2007	writer,director,actor	tt0050986,tt0060827,tt0069467,tt0050976
6	nm0000006	Ingrid Bergman	1915	1982	actress,soundtrack,producer	tt0077711,tt0038109,tt0034583,tt0036855
7	nm0000007	Humphrey Bogart	1899	1957	actor,soundtrack,producer	tt0043265,tt0034583,tt0042593,tt0037382
8	nm0000008	Marlon Brando	1924	2004	actor,soundtrack,director	tt0078788,tt0068646,tt0070849,tt0047296
9	nm0000009	Richard Burton	1925	1984	actor,soundtrack,producer	tt0061184,tt0087803,tt0057877,tt0059749
10	nm0000010	James Cagney	1899	1986	actor,soundtrack,director	tt0029870,tt0035575,tt0042041,tt0055256

- There
  - Is one composite attribute, primaryName.
  - Are two multivalued attributes: primaryProfession, knownForTitles
  - knownForTitles is also tricky, which we will see.
  - Names are also a little tricky

Aggregation

Aggregation

- Consider the ternary relationship *proj_guide*, which we saw earlier
- Suppose we want to record evaluations of a student by a guide on a project



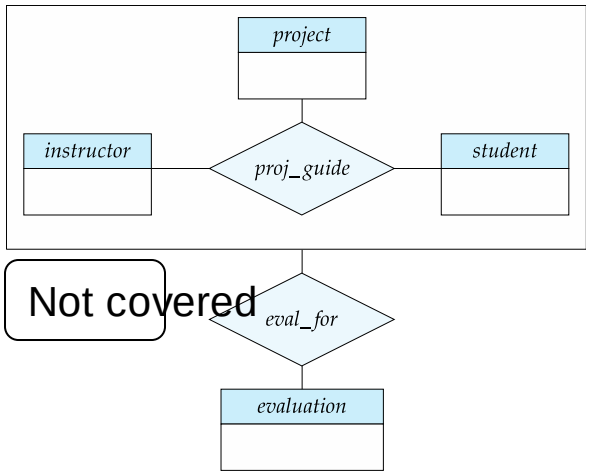
Aggregation (Cont.)

- Relationship sets *eval_for* and *proj_guide* represent overlapping information
  - Every *eval_for* relationship corresponds to a *proj_guide* relationship
  - However, some *proj_guide* relationships may not correspond to any *eval_for* relationships
    - So we can't discard the *proj_guide* relationship
- Eliminate this redundancy via *aggregation*
  - Treat relationship as an abstract entity
  - Allows relationships between relationships
  - Abstraction of relationship into new entity

Not covered

Aggregation (Cont.)

- Eliminate this redundancy via *aggregation* without introducing redundancy, the following diagram represents:
  - A student is guided by a particular instructor on a particular project
  - A student, instructor, project combination may have an associated evaluation



The simplest way to handle in relational is an associative entity.

Some thoughts here:
https://www.geeksforgeeks.org/aggregate-data-model-in-nosql/

Reduction to Relational Schemas

- To represent aggregation, create a schema containing
  - Primary key of the aggregated relationship,
  - The primary key of the associated entity set
  - Any descriptive attributes
- In our example:
  - The schema *eval_for* is:

    eval_for (*s_ID, project_id, i_ID, evaluation_id*)
  - The schema *proj_guide* is redundant.

Not covered

Let’s Examine and Do Some Examples

Students, Faculty, Person

Relational Model

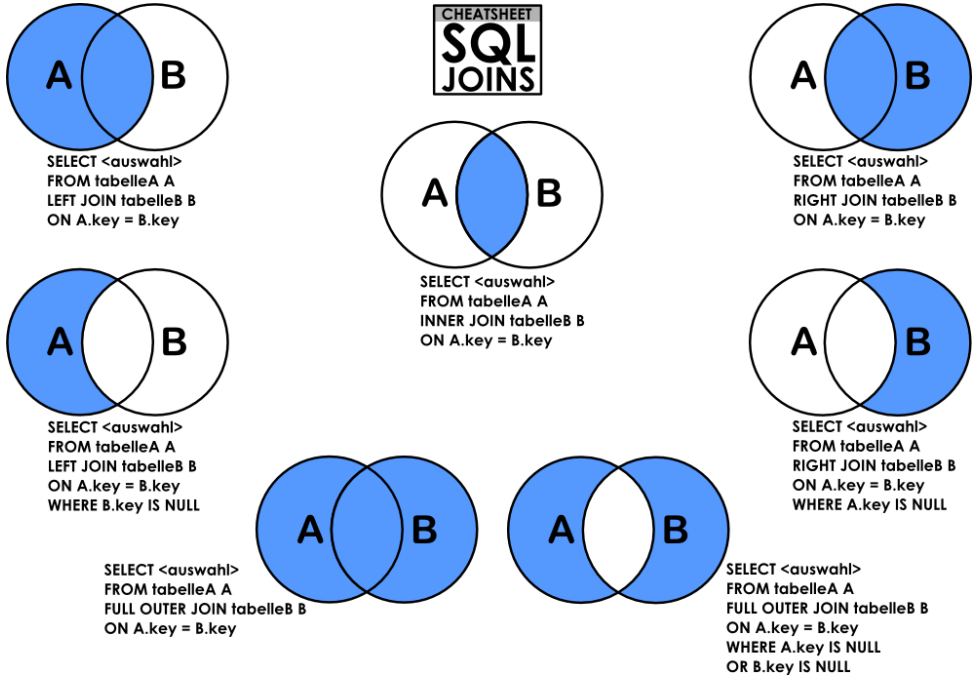
What are all those other Symbols?

- τ order by
- γ group by
- $\neg$ negation
- $\div$ set division
- $\bowtie$ natural join, theta-join
- $\ltimes$ left outer join
- $\rtimes$ right outer join
- $\ltimes\rtimes$ full outer join
- $\ltimes$ left semi join
- $\rtimes$ right semi join
- $\triangleright$ 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 (⟕, ⟖)



What are all those other Symbols?

- τ order by
- γ group by
- ¬ negation
- ÷ 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.

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*

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

SQL

Set Operations

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)

Note:

- Show in Relax calculator.
- Not all databases support INTERSECT and EXCEPT
- We will see how to handle below.

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**.

Subqueries
Concepts and Examples
(Including Set Membership)

Nested Subqueries

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

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

  as follows:

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

    B <operation> (subquery)

    B is an attribute and <operation> to be defined later.
  - **Select clause:**

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

Covered. Review.

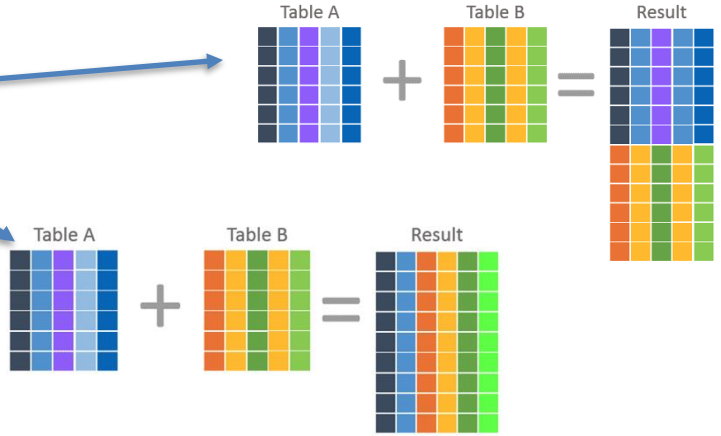
Note:

- 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.

Covered. Review.



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
8988	Tanaka	Biology	120

Covered. Review.

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)
```

Covered. Review.

Student

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

Nested Subqueries

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

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

  as follows:

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

    B <operation> (subquery)

    B is an attribute and <operation> to be defined later.
  - **Select clause:**

    A_i can be replaced be 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.

Covered. Review.

Subqueries
Concepts and Examples
(Including 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 <comp> **some** $r \Leftrightarrow \exists\, t \in r$ such that (F <comp> t)
  Where <comp> can be: $<,\ \leq,\ >,\ =,\ \neq$

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

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

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

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

(= **some**) ≡ **in**
However, (≠ **some**) ≢ **not in**

Set Comparison – “all” Clause

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

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

Definition of "all" Clause

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

(5 < **all** {0, 5, 6}) = false

(5 < **all** {6, 10}) = true

(5 = **all** {4, 5}) = false

(5 ≠ **all** {4, 6}) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**

However, (= **all**) ≢ **in**

Test for Empty Relations

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

Use of “exists” Clause

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

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

- **Correlation name** – variable S in the outer query
- **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

Some Types and Functions

Functions

MySQL CHEAT SHEET: STRING FUNCTIONS

by sqlbackupandftp.com with ♥

MEASUREMENT

Return a string containing binary representation of a number

```
BIN (12) = '1100'
```

Return length of argument in bits

```
BIT_LENGTH ('MySql') = 40
```

Return number of characters in argument

```
CHAR_LENGTH ('MySql') = 5
CHARACTER_LENGTH ('MySql') = 5
```

Return the length of a string in bytes

```
LENGTH ('Ö') = 2
LENGTH ('A') = 1
OCTET_LENGTH ('Ö') = 2
OCTET_LENGTH ('X') = 1
```

Return a soundex string

```
SOUNDEX ('MySql') = 'M240'
SOUNDEX ('MySqlDatabase') = 'M24312'
```

Compare two strings

```
STRCMP ('A', 'A') = 0
STRCMP ('A', 'B') = -1
STRCMP ('B', 'A') = 1
```

SEARCH

Return the index of the first occurrence of substring

```
INSTR ('MySql', 'Sql') = 3
INSTR ('Sql', 'MySql') = 0
```

Return the position of the first occurrence of substring

```
LOCATE ('Sql', 'MySqlSql') = 3
LOCATE ('xSql', 'MySql') = 0
LOCATE ('Sql', 'MySqlSql', 5) = 6
POSITION('Sql' IN 'MySqlSql') = 3
```

Pattern matching using regular expressions

```
'abc' RLIKE '[a-z]+' = 1
'123' RLIKE '[a-z]+' = 0
```

Return a substring from a string before the specified number of occurrences of the delimiter

```
SUBSTRING_INDEX ('A:B:C', ':', 1) = 'A'
SUBSTRING_INDEX ('A:B:C', ':', 2) = 'A:B'
SUBSTRING_INDEX ('A:B:C', ':', -2) = 'B:C'
```

CONVERSION

Return numeric value of left-most character

```
ASCII ('2') = 50
ASCII (2) = 50
ASCII ('dx') = 100
```

Return the character for each number passed

```
CHAR (77.3,121,83,81, '76, 81.6') = 'MySQL'
CHAR (45*256+45) = CHAR (45,45) = '--'
CHARSET(CHAR (X'65' USING utf8)) = 'utf8'
```

Decode to / from a base-64 string

```
TO_BASE64 ('abc') = 'YWJj'
FROM_BASE64 ('YWJj') = 'abc'
```

Convert string or number to its hexadecimal representation

```
X'616263' = 'abc'
HEX ('abc') = 616263
HEX(255) = 'FF'
CONV(HEX(255), 16, 10) = 255
```

Convert each pair of hexadecimal digits to a character

```
UNHEX ('4D7953514C') = 'MySQL'
UNHEX ('GG') = NULL
UNHEX (HEX ('abc')) = 'abc'
```

Return the argument in lowercase

```
LOWER ('MYSQL') = 'mysql'
LCASE ('MYSQL') = 'mysql'
```

Load the named file

```
SET blob_col=LOAD_FILE ('/tmp/picture')
```

Return a string containing octal representation of a number

```
OCT (12) = '14'
```

Return character code for leftmost character of the argument

```
ORD ('2') = 50
```

Escape the argument for use in an SQL statement

```
QUOTE ('Don\'t!') = 'Don\'t!'
QUOTE (NULL) = 'NULL'
```

Convert to uppercase

```
UPPER ('mysql') = 'MYSQL'
UCASE ('mysql') = 'MYSQL'
```

MODIFICATION

Return concatenated string

```
CONCAT ('My', 'S', 'QL') = 'MySQL'
CONCAT ('My', NULL, 'QL') = NULL
CONCAT (14.3) = '14.3'
```

Return concatenate with separator

```
CONCAT_WS (',', 'My', 'Sql') = 'My,Sql'
CONCAT_WS (',','My',NULL,'Sql') = 'My,Sql'
```

Return a number formatted to specified number of decimal places

```
FORMAT (12332.123456, 4) = 12,332.1235
FORMAT (12332.1, 4) = 12,332.1000
FORMAT (12332.2, 0) = 12332.2
FORMAT (12332.2, 2, 'de_DE') = 12.332,20
```

Insert a substring at the specified position up to the specified number of characters

```
INSERT ('12345', 3, 2, 'ABC') = '12ABC5'
INSERT ('12345', 10, 2, 'ABC') = '12345'
INSERT ('12345', 3, 10, 'ABC') = '12ABC'
```

Return the leftmost number of characters as specified

```
LEFT ('MySql', 2) = 'My'
```

Return the string argument, left-padded with the specified string

```
LPAD ('Sql', 2, ':)') = 'Sq'
LPAD ('Sql', 4, ':)') = ':Sql'
LPAD ('Sql', 7, ':)') = ':):)Sql'
```

Remove leading spaces

```
LTRIM ('      MySql') = 'MySql'
```

Repeat a string the specified number of times

```
REPEAT ('MySQL', 3) = 'MySQLMySQLMySQL'
```

Replace occurrences of a specified string

```
REPLACE ('NoSql', 'No', 'My') = 'MySql'
```

Reverse the characters in a string

```
REVERSE ('MySql') = 'lqSyM'
```

Return the specified rightmost number of characters

```
RIGHT ('MySql', 3) = 'Sql'
```

Returns the string argument, right-padded with the specified strin.

```
RPAD ('Sql', 2, ':)') = 'Sq'
RPAD ('Sql', 4, ':)') = 'Sql:'
RPAD ('Sql', 7, ':)') = 'Sql:):)'
```

Remove trailing spaces

```
RTRIM ('MySql      ') = 'MySql'
```

Return a string of the specified number of spaces

```
SPACE ('6') = '      '
```

Return the substring as specified

```
SUBSTRING=SUBSTR=MID('MySql',3) = 'Sql'
SUBSTRING=SUBSTR=MID('MySql' FROM 4) = 'ql'
SUBSTRING=SUBSTR=MID('MySql',3,1) = 'S'
SUBSTRING=SUBSTR=MID('MySql',-3) = 'Sql'
SUBSTRING=SUBSTR=MID('MySql' FROM -4 FOR 2)
= 'yS'
```

Remove leading and trailing spaces

```
TRIM(' MySql ') = 'MySql'
TRIM(LEADING 'x' FROM 'xxxSqlMy') = 'MySql'
TRIM(BOTH 'My' FROM 'MySqlMy') = 'Sql'
TRIM(TRAILING 'Sql' FROM 'MySql') = 'My'
```

SETS

Return string at index number

```
ELT (1, 'ej', 'Heja', 'hej', 'foo') = 'ej'
ELT (4, 'ej', 'Heja', 'hej', 'foo') = 'foo'
```

Return a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string

```
EXPORT_SET (5,'Y','N',',',4) = 'Y,N,Y,N'
EXPORT_SET (6,'1','0',',',6) = '0,1,1,0,0,0'
```

Return the index (position) of the first argument in the subsequent arguments

```
FIELD ('ej','Hj','ej','Heja','hej','oo') = 2
FIELD ('fo','Hj','ej','Heja','hej','oo') = 0
```

Return the index position of the first argument within the second argument

```
FIND_IN_SET ('b', 'a,b,c,d') = 2
FIND_IN_SET ('z', 'a,b,c,d') = 0
FIND_IN_SET ('a,', 'a,b,c,d') = 0
```

Return a set of comma-separated strings that have the corresponding bit in bits set

```
MAKE_SET (1,'a','b','c') = 'a'
MAKE_SET (1|4,'ab','cd','ef') = 'ab,ef'
MAKE_SET (1|4,'ab','cd',NULL,'ef') = 'ab'
MAKE_SET (0,'a','b','c') = ''
```

Standard SQL Functions Cheat Sheet

LearnSQL.com

TEXT FUNCTIONS

CONCATENATION

Use the `||` operator to concatenate two strings:

```
SELECT 'Hi ' || 'there!';
-- result: Hi there!
```

Remember that you can concatenate only character strings using `||`. Use this trick for numbers:

```
SELECT '' || 4 || 2;
-- result: 42
```

Some databases implement non-standard solutions for concatenating strings like `CONCAT()` or `CONCAT_WS()`. Check the documentation for your specific database.

LIKE OPERATOR – PATTERN MATCHING

Use the `_` character to replace any single character. Use the `%` character to replace any number of characters (including 0 characters).

Fetch all names that start with any letter followed by `'atherine'`:

```
SELECT name
FROM names
WHERE name LIKE '_atherine';
```

Fetch all names that end with `'a'`:

```
SELECT name
FROM names
WHERE name LIKE '%a';
```

USEFUL FUNCTIONS

Get the count of characters in a string:

```
SELECT LENGTH('LearnSQL.com');
-- result: 12
```

Convert all letters to lowercase:

```
SELECT LOWER('LEARNSQL.COM');
-- result: learnsql.com
```

Convert all letters to uppercase:

```
SELECT UPPER('LearnSQL.com');
-- result: LEARNSQL.COM
```

Convert all letters to lowercase and all first letters to uppercase (not implemented in MySQL and SQL Server):

```
SELECT INITCAP('edgar frank ted cODD');
-- result: Edgar Frank Ted Codd
```

Get just a part of a string:

```
SELECT SUBSTRING('LearnSQL.com', 9);
-- result: .com
SELECT SUBSTRING('LearnSQL.com', 0, 6);
-- result: Learn
```

Replace part of a string:

```
SELECT REPLACE('LearnSQL.com', 'SQL',
'Python');
-- result: LearnPython.com
```

NUMERIC FUNCTIONS

BASIC OPERATIONS

Use `+`, `-`, `*`, `/` to do some basic math. To get the number of seconds in a week:

```
SELECT 60 * 60 * 24 * 7; -- result: 604800
```

CASTING

From time to time, you need to change the type of a number. The `CAST()` function is there to help you out. It lets you change the type of value to almost anything (`integer`, `numeric`, `double precision`, `varchar`, and many more).

Get the number as an integer (without rounding):

```
SELECT CAST(1234.567 AS integer);
-- result: 1234
```

Change a column type to `double precision`

```
SELECT CAST(column AS double precision);
```

USEFUL FUNCTIONS

Get the remainder of a division:

```
SELECT MOD(13, 2);
-- result: 1
```

Round a number to its nearest integer:

```
SELECT ROUND(1234.56789);
-- result: 1235
```

Round a number to three decimal places:

```
SELECT ROUND(1234.56789, 3);
-- result: 1234.568
```

PostgreSQL requires the first argument to be of the type `numeric` – cast the number when needed.

To round the number **up**:

```
SELECT CEIL(13.1); -- result: 14
SELECT CEIL(-13.9); -- result: -13
```

The `CEIL(x)` function returns the **smallest** integer **not less** than `x`. In SQL Server, the function is called `CEILING()`.

To round the number **down**:

```
SELECT FLOOR(13.8); -- result: 13
SELECT FLOOR(-13.2); -- result: -14
```

The `FLOOR(x)` function returns the **greatest** integer **not greater** than `x`.

To round towards 0 irrespective of the sign of a number:

```
SELECT TRUNC(13.5); -- result: 13
SELECT TRUNC(-13.5); -- result: -13
```

`TRUNC(x)` works the same way as `CAST(x AS integer)`. In MySQL, the function is called `TRUNCATE()`.

To get the absolute value of a number:

```
SELECT ABS(-12); -- result: 12
```

To get the square root of a number:

```
SELECT SQRT(9); -- result: 3
```

NULLs

To retrieve all rows with a missing value in the `price` column:

```
WHERE price IS NULL
```

To retrieve all rows with the `weight` column populated:

```
WHERE weight IS NOT NULL
```

Why shouldn't you use `price = NULL` or `weight != NULL`? Because databases don't know if those expressions are true or false – they are evaluated as `NULL`s.

Moreover, if you use a function or concatenation on a column that is `NULL` in some rows, then it will get propagated. Take a look:

domain	LENGTH(domain)
LearnSQL.com	12
LearnPython.com	15
NULL	*NULL*
vertabelo.com	13

USEFUL FUNCTIONS

COALESCE(x, y, ...)

To replace `NULL` in a query with something meaningful:

```
SELECT
  domain,
  COALESCE(domain, 'domain missing')
FROM contacts;
```

domain	coalesce
LearnSQL.com	LearnSQL.com
NULL	domain missing

The `COALESCE()` function takes any number of arguments and returns the value of the first argument that isn't `NULL`.

NULLIF(x, y)

To save yourself from *division by 0* errors:

```
SELECT
  last_month,
  this_month,
  this_month * 100.0
    / NULLIF(last_month, 0)
    AS better_by_percent
FROM video_views;
```

last_month	this_month	better_by_percent
723786	1085679	150.0
0	178123	*NULL*

The `NULLIF(x, y)` function will return `NULL` if `x` is the same as `y`, else it will return the `x` value.

CASE WHEN

The basic version of `CASE WHEN` checks if the values are equal (e.g., if `fee` is equal to 50, then `'normal'` is returned). If there isn't a matching value in the `CASE WHEN`, then the `ELSE` value will be returned (e.g., if `fee` is equal to 49, then `'not available'` will show up.

```
SELECT
  CASE fee
    WHEN 50 THEN 'normal'
    WHEN 10 THEN 'reduced'
    WHEN  0 THEN 'free'
    ELSE 'not available'
  END AS tariff
FROM ticket_types;
```

The most popular type is the **searched CASE WHEN** – it lets you pass conditions (as you'd write them in the `WHERE` clause), evaluates them in order, then returns the value for the first condition met.

```
SELECT
  CASE
    WHEN score >= 90 THEN 'A'
    WHEN score >  60 THEN 'B'
    ELSE 'F'
  END AS grade
FROM test_results;
```

Here, all students who scored at least 90 will get an A, those with the score above 60 (and below 90) will get a B, and the rest will receive an F.

TROUBLESHOOTING

Integer division

When you don't see the decimal places you expect, it means that you are dividing between two integers. Cast one to `decimal`:

```
CAST(123 AS decimal) / 2
```

Division by 0

To avoid this error, make sure that the denominator is not equal to 0. You can use the `NULLIF()` function to replace 0 with a `NULL`, which will result in a `NULL` for the whole expression:

```
count / NULLIF(count_all, 0)
```

Inexact calculations

If you do calculations using `real` (floating point) numbers, you'll end up with some inaccuracies. This is because this type is meant for scientific calculations such as calculating the velocity. Whenever you need accuracy (such as dealing with monetary values), use the `decimal` / `numeric` type (or money if available).

Errors when rounding with a specified precision

Most databases won't complain, but do check the documentation if they do. For example, if you want to specify the rounding precision in PostgreSQL, the value must be of the `numeric` type.

Try out the interactive **Standard SQL Functions** course at **LearnSQL.com**, and check out our other SQL courses.

Standard SQL Functions Cheat Sheet

LearnSQL.com

AGGREGATION AND GROUPING

- **COUNT(expr)** – the count of values for the rows within the group
- **SUM(expr)** – the sum of values within the group
- **AVG(expr)** – the average value for the rows within the group
- **MIN(expr)** – the minimum value within the group
- **MAX(expr)** – the maximum value within the group

To get the number of rows in the table:

```
SELECT COUNT(*)
FROM city;
```

To get the number of non-NULL values in a column:

```
SELECT COUNT(rating)
FROM city;
```

To get the count of unique values in a column:

```
SELECT COUNT(DISTINCT country_id)
FROM city;
```

GROUP BY

CITY	
name	**country_id**
Paris	1
Marseille	1
Lyon	1
Berlin	2
Hamburg	2
Munich	2
Warsaw	4
Cracow	4

→

CITY	
country_id	**count**
1	3
2	3
4	2

The example above – the count of cities in each country:

```
SELECT name, COUNT(country_id)
FROM city
GROUP BY name;
```

The average rating for the city:

```
SELECT city_id, AVG(rating)
FROM ratings
GROUP BY city_id;
```

Common mistake: COUNT(*) and LEFT JOIN

When you join the tables like this: `client LEFT JOIN project`, and you want to get the number of projects for every client you know, `COUNT(*)` will return **1** for each client even if you've never worked for them. This is because, they're still present in the list but with the `NULL` in the fields related to the project after the `JOIN`. To get the correct count (**0** for the clients you've never worked for), count the values in a column of the other table, e.g., `COUNT(project_name)`. Check out this exercise to see an example.

DATE AND TIME

There are 3 main time-related types: `date`, `time`, and `timestamp`. Time is expressed using a 24-hour clock, and it can be as vague as just hour and minutes (e.g., `15:30` – 3:30 p.m.) or as precise as microseconds and time zone (as shown below):

```
2021-12-31 14:39:53.662522-05
   date    time
      timestamp
YYYY-mm-dd HH:MM:SS.ssssss±TZ
```

`14:39:53.662522-05` is almost 2:40 p.m. CDT (e.g., in Chicago; in UTC it'd be 7:40 p.m.). The letters in the above example represent:

In the date part:

- YYYY – the 4-digit year.
- mm – the zero-padded month (01—January through 12—December).
- dd – the zero-padded day.

In the time part:

- HH – the zero-padded hour in a 24-hour clock.
- MM – the minutes.
- SS – the seconds. *Omissible.*
- ssssss – the smaller parts of a second – they can be expressed using 1 to 6 digits. *Omissible.*
- ±TZ – the timezone. It must start with either + or −, and use two digits relative to UTC. *Omissible.*

What time is it?

To answer that question in SQL, you can use:

- CURRENT_TIME – to find what time it is.
- CURRENT_DATE – to get today's date. (`GETDATE()` in SQL Server.)
- CURRENT_TIMESTAMP – to get the timestamp with the two above.

Creating values

To create a `date`, `time`, or `timestamp`, simply write the value as a string and cast it to the proper type.

```
SELECT CAST('2021-12-31' AS date);
SELECT CAST('15:31' AS time);
SELECT CAST('2021-12-31 23:59:29+02' AS
timestamp);
SELECT CAST('15:31.124769' AS time);
```

Be careful with the last example – it will be interpreted as 15 minutes 31 seconds and 124769 microseconds! It is always a good idea to write `00` explicitly for hours: `'00:15:31.124769'`.

You might skip casting in simple conditions – the database will know what you mean.

```
SELECT airline, flight_number, departure_time
FROM airport_schedule
WHERE departure_time < '12:00';
```

INTERVALs

Note: In SQL Server, intervals aren't implemented – use the `DATEADD()` and `DATEDIFF()` functions.

To get the simplest interval, subtract one time value from another:

```
SELECT CAST('2021-12-31 23:59:59' AS
timestamp) - CAST('2021-06-01 12:00:00' AS
timestamp);
-- result: 213 days 11:59:59
```

To define an interval: `INTERVAL '1' DAY`

This syntax consists of three elements: the `INTERVAL` keyword, a quoted value, and a time part keyword (in singular form.) You can use the following time parts: `YEAR`, `MONTH`, `WEEK`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`. In MySQL, omit the quotes. You can join many different `INTERVAL`s using the `+` or `-` operator:

```
INTERVAL '1' YEAR + INTERVAL '3' MONTH
```

In some databases, there's an easier way to get the above value. And it accepts plural forms! `INTERVAL '1 year 3 months'`

There are two more syntaxes in the Standard SQL:

Syntax	What it does
`INTERVAL 'x-y' YEAR TO MONTH`	`INTERVAL 'x year y month'`
`INTERVAL 'x-y' DAY TO SECOND`	`INTERVAL 'x day y second'`

In MySQL, write `year_month` instead of `YEAR TO MONTH` and `day_second` instead of `DAY TO SECOND`.

To get the last day of a month, add one month and subtract one day:

```
SELECT CAST('2021-02-01' AS date)
       + INTERVAL '1' MONTH
       - INTERVAL '1' DAY;
```

To get all events for next three months from today:

```
SELECT event_date, event_name
FROM calendar
WHERE event_date BETWEEN CURRENT_DATE AND
CURRENT_DATE + INTERVAL '3' MONTH;
```

To get part of the date:

```
SELECT EXTRACT(YEAR FROM birthday)
FROM artists;
```

One of possible returned values: 1946. In SQL Server, use the `DATEPART(part, date)` function.

TIME ZONES

In the SQL Standard, the `date` type can't have an associated time zone, but the `time` and `timestamp` types can. In the real world, time zones have little meaning without the date, as the offset can vary through the year because of **daylight saving time**. So, it's best to work with the `timestamp` values.

When working with the type `timestamp with time zone` (abbr. `timestamptz`), you can type in the value in your local time zone, and it'll get converted to the UTC time zone as it is inserted into the table. Later when you select from the table it gets converted back to your local time zone. This is immune to time zone changes.

AT TIME ZONE

To operate between different time zones, use the `AT TIME ZONE` keyword.

If you use this format: `{timestamp without time zone} AT TIME ZONE {time zone}`, then the database will read the time stamp in the specified time zone and convert it to the time zone local to the display. It returns the time in the format `timestamp with time zone`.

If you use this format: `{timestamp with time zone} AT TIME ZONE {time zone}`, then the database will convert the time in one time zone to the target time zone specified by `AT TIME ZONE`. It returns the time in the format `timestamp without time zone`, in the target time zone.

You can define the time zone with popular shortcuts like UTC, `MST`, or `GMT`, or by `continent/city` such as: `America/New_York`, `Europe/London`, and `Asia/Tokyo`.

Examples

We set the local time zone to `'America/New_York'`.

```
SELECT TIMESTAMP '2021-07-16 21:00:00' AT
TIME ZONE 'America/Los_Angeles';
-- result: 2021-07-17 00:00:00-04
```

Here, the database takes a timestamp without a time zone and it's told it's in Los Angeles time, which is then converted to the local time – New York for displaying. This answers the question **"At what time should I turn on the TV if the show starts at 9 PM in Los Angeles?"**

```
SELECT TIMESTAMP WITH TIME ZONE '2021-06-20
19:30:00' AT TIME ZONE 'Australia/Sydney';
-- result: 2021-06-21 09:30:00
```

Here, the database gets a timestamp specified in the local time zone and converts it to the time in Sydney (note that it didn't return a time zone.) This answers the question **"What time is it in Sydney if it's 7:30 PM here?"**

Functions

- The are dozens if not hundreds of standard functions in SQL.
- All DBMS implementations have product specific functions.
- General rule:
  - If you have to do something, ask yourself
  - Am I the first one who ever had to do this?
  - If the answer is “No,” then ask Dr. Google/ChatGPT.
  - If the answer is Yes,” ask yourself, “Am I sure this is a good idea.”
- The functions are useful and straightforward.
- Some examples ➔ To the notebook we go, yo ho!

Views

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  select *ID*, *name*, *dept_name*
  from *instructor*

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.

View Definition

- A view is defined using the **create view** statement which has the form

  create view *v* **as** < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

View Definition and Use

- A view of instructors without their salary

```
create view faculty as
    select ID, name, dept_name
    from instructor
```

- Find all instructors in the Biology department

```
select name
from faculty
where dept_name = 'Biology'
```

- Create a view of department salary totals

```
create view departments_total_salary(dept_name, total_salary) as
    select dept_name, sum (salary)
    from instructor
    group by dept_name;
```

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to ***depend directly*** on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to ***depend on*** view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be ***recursive*** if it depends on itself.

Views Defined Using Other Views

- **create view** ***physics_fall_2017*** **as**
  select *course.course_id*, *sec_id*, *building*, *room_number*
  from *course*, *section*
  where *course.course_id* = *section.course_id*
  and *course.dept_name* = 'Physics'
  and *section.semester* = 'Fall'
  and *section.year* = '2017';

- **create view** ***physics_fall_2017***_*watson* **as**
  select *course_id*, *room_number*
  from ***physics_fall_2017***
  where *building*= 'Watson';

View Expansion

- Expand the view :

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from physics_fall_2017
    where building= 'Watson'
```

- To:

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from (select course.course_id, building, room_number
        from course, section
        where course.course_id = section.course_id
            and course.dept_name = 'Physics'
            and section.semester = 'Fall'
            and section.year = '2017')
    where building= 'Watson';
```

View Expansion (Cont.)

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

  repeat
  Find any view relation v_i in e_1
  Replace the view relation v_i by the expression defining v_i
  until no more view relations are present in e_1

- As long as the view definitions are not recursive, this loop will terminate

Materialized Views

- Certain database systems allow view relations to be physically stored.
  - Physical copy created when the view is defined.
  - Such views are called **Materialized view**:
- If relations used in the query are updated, the materialized view result becomes out of date
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated.

Update of a View

- Add a new tuple to *faculty* view which we defined earlier

  insert into *faculty*

  values ('30765', 'Green', 'Music');

- This insertion must be represented by the insertion into the *instructor* relation
  - Must have a value for salary.
- Two approaches
  - Reject the insert
  - Inset the tuple

    ('30765', 'Green', 'Music', null)

    into the *instructor* relation

Some Updates Cannot be Translated Uniquely

- **create view** *instructor_info* **as**
  select *ID*, *name*, *building*
  from *instructor*, *department*
  where *instructor.dept_name= department.dept_name*;
- **insert into** *instructor_info*
  values ('69987', 'White', 'Taylor');
- Issues
  - Which department, if multiple departments in Taylor?
  - What if no department is in Taylor?

And Some Not at All

- **create view** *history_instructors* **as**
  select *
  from *instructor*
  where *dept_name*= 'History';
- What happens if we insert

  ('25566', 'Brown', 'Biology', 100000)

  into *history_instructors?*

View Updates in SQL

- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation.
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group** by or **having** clause.

Codd's 12 Rules
Metadata

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.

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.

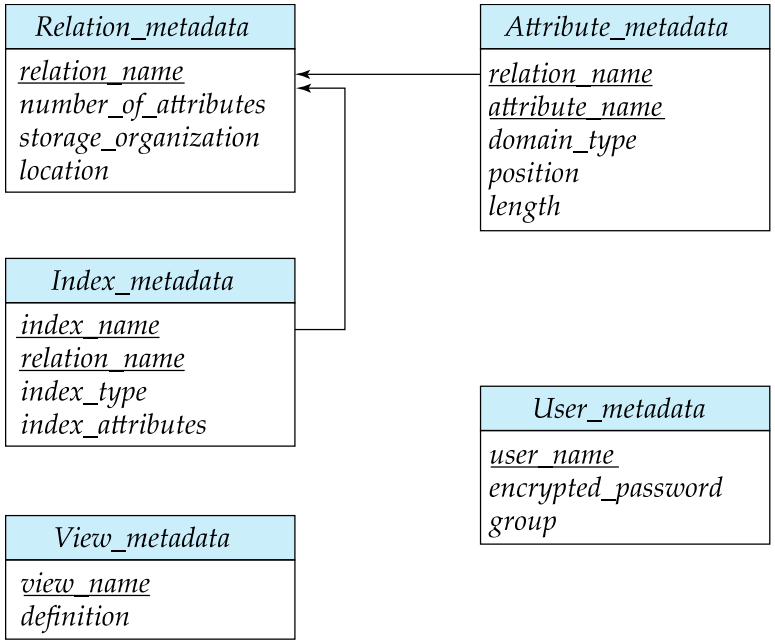
Data Dictionary Storage

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

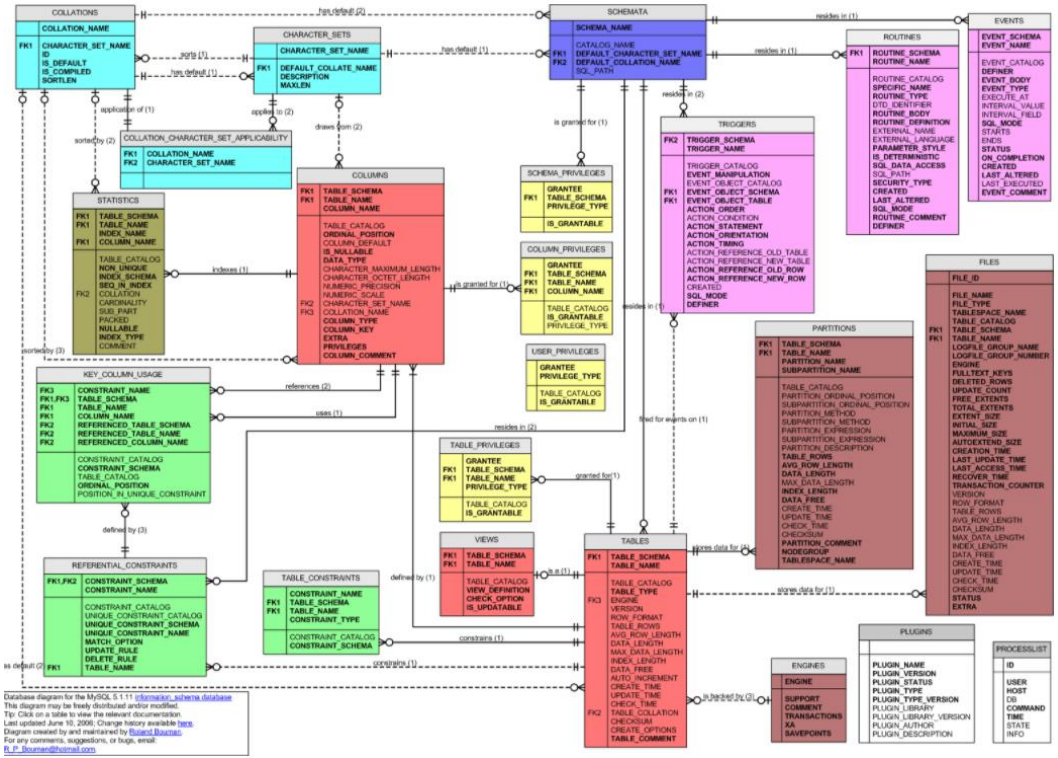
- Information about relations
  - names of relations
  - names, types and lengths of attributes of each relation
  - names and definitions of views
  - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
  - number of tuples in each relation
- 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.

Do Some Examples if not too Tired or Bored

REST

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.

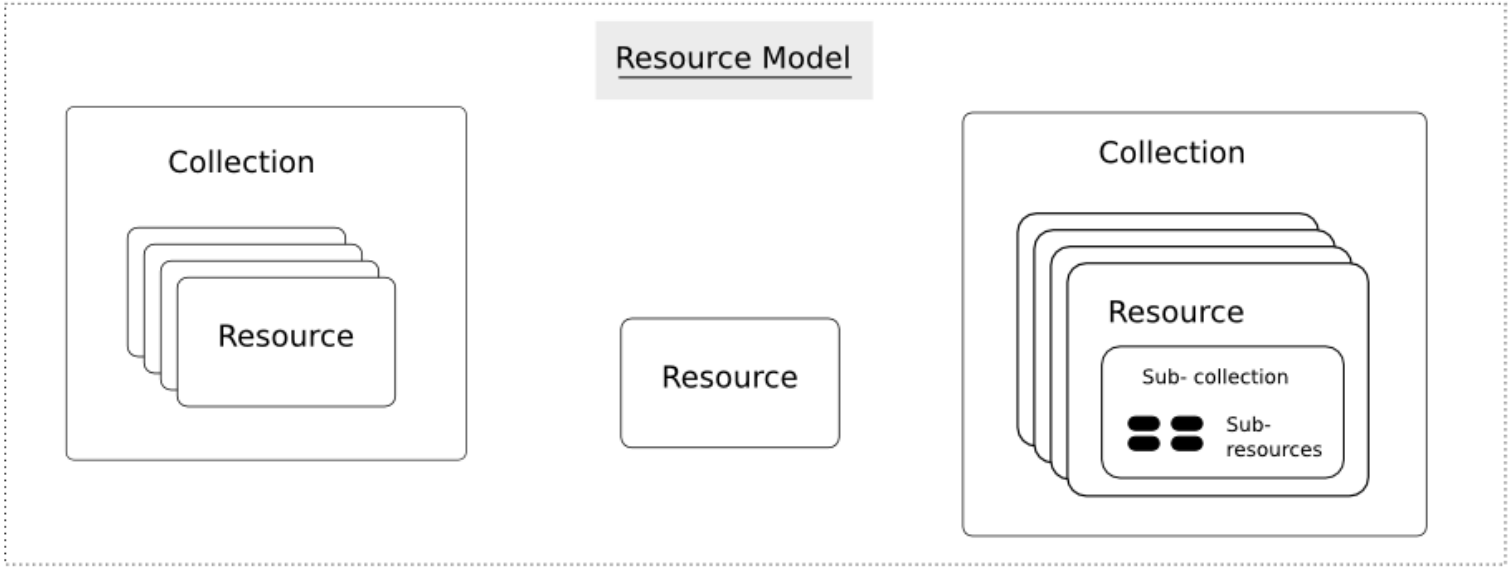
Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP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

Creating RESTFul Webservice

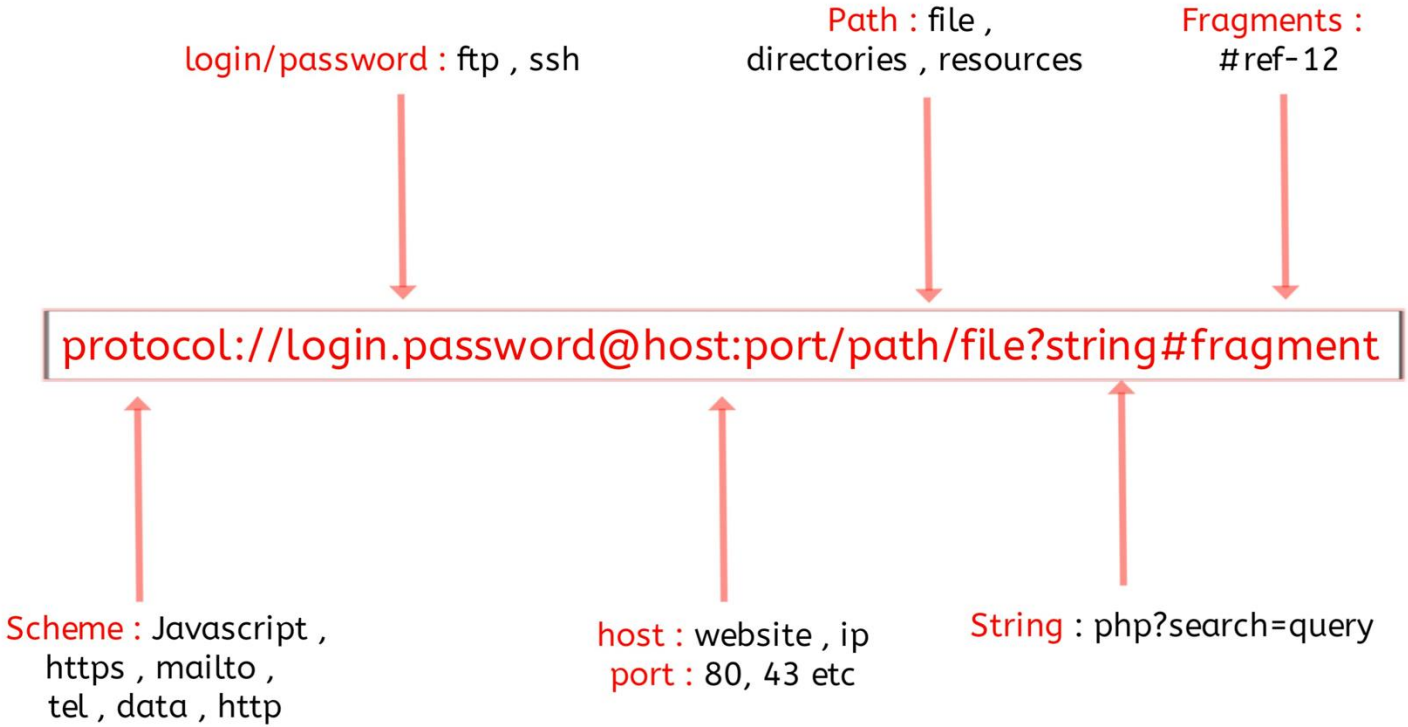
In next chapters, we'll create a webservice say user management with following functionalities –

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



URLs



jdbc:mysql://columbia-examples.ckkqqktwkcji.us-east-1.rds.amazonaws.com:3306

GET http://localhost:5001/f23_imdb_clean/name_basics/nm0000158

GET http://localhost:5001/f23_imdb_clean/name_basics?deathYear=2023&birthyear=1960

```
select * from f23_imdb_clean.name_basics where
    deathYear=2023 AND birthyear=1960
```

PUT http://localhost:5001/f23_imdb_clean/name_basics ?deathYear=2023&birthyear=1960

Body {'primaryName': 'Does not matter cause is dead.'}

```
update f23_imdb_clean.name_basics
    set
    where deathYear=2023 AND birthyear=1960
```

Simplistic, Conceptual Mapping (Examples)

REST Method	Resource Path	Relational Operation	DB Resource
DELETE	/people	DROP TABLE	people table
POST	/people	INSERT INTO PEOPLE (...) VALUES(...)	people table people row
GET	/people/21	SHOW KEYS FROM people ...; SELECT * FROM people WHERE playerID= 21	people row
GET	/people/21/batting	SELECT batting.* FROM people JOIN batting USING(playerID) WHERE playerID=21	
GET	/people/21/batting/2004_1	SELECT batting.* FROM people JOIN batting USING(playerID) WHERE playerID=21 AND yearID=2004 AND stint=1	

Application Architecture

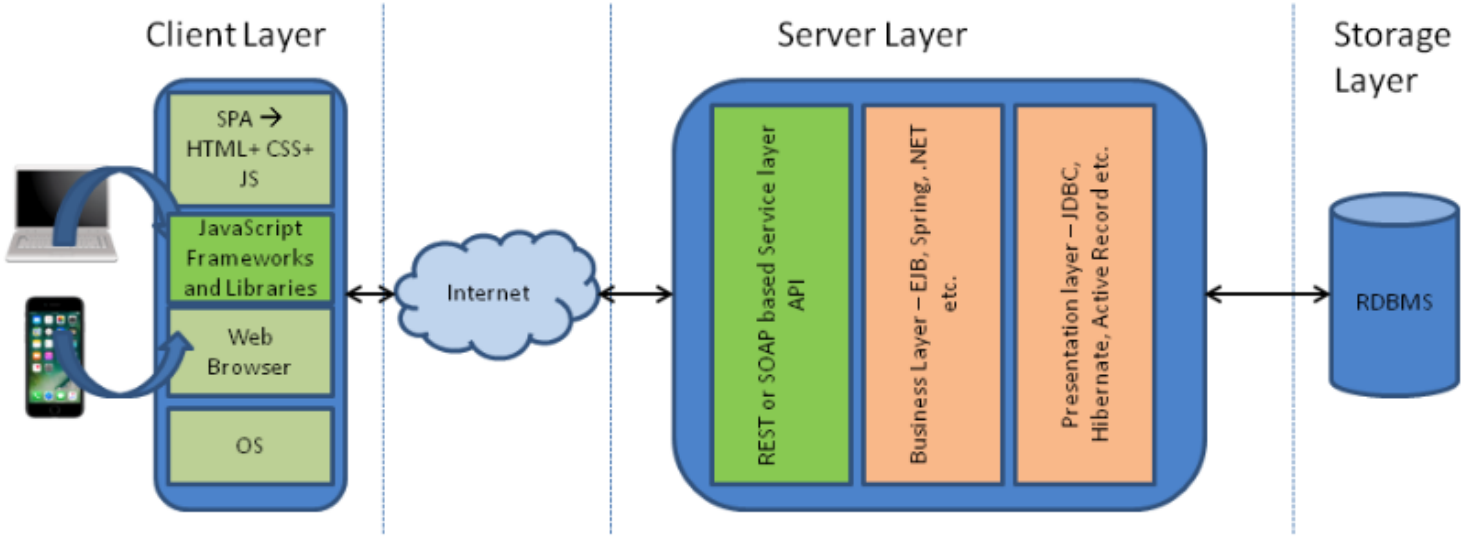


Diagram 2: The moving of the Web Layer from the Server to the Client