

*W4111 – Introduction to Databases*  
*Section 002, Spring 2025*  
*Lecture 5: ER(4), Relational(4), SQL(4)*



# *Today's Contents*

# Contents

- Introduction and course updates
- Some of Codd's 12 Rules: Information Rule, Online Catalog, View Update
- ER Modeling:
  - Degree, Cardinality
  - Aggregation
- Relational Algebra/Model
  - Review of joins
  - Obscure operators: anti-join, semi-join, group by, order by, division
- SQL
  - Some datatypes and functions – quick enumeration
  - Indexes
  - Functions, Procedures, Triggers
  - Introduction to REST and web applications



# Introduction and course updates

# Course Updates

- Exam 1
  - Makeup
    - I posted instructions for makeup exam on Ed.
    - You must take the exam *in person*.
    - I strongly encourage people to take the exam on the afternoon of Sunday, 12-October when I am on campus. I will publish the exact time as soon as I know.
    - If you cannot take the exam on Sunday, you must arrange to take the exam with a TA in person in office hours. You are responsible for arranging with a TA.
    - Your deadline is the start of class on 17-Oct.
  - Grading: We are grading this weekend and early next week.
- Homework 3 will come out the weekend of 18-October.  
You will have two weeks to complete.

# *Codd's 12 Rules*

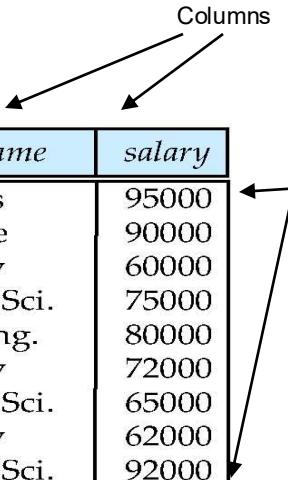


# Relational Model

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



**Ted Codd**  
Turing Award 1981



<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
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

The Database Explorer shows the following tree structure under the 'localhost' connection:

- @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

- 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, ... ....

22	def	information_schema	RESOURCE_GROUPS
23	def	information_schema	ROUTINES
			... omitted ...



# 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](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

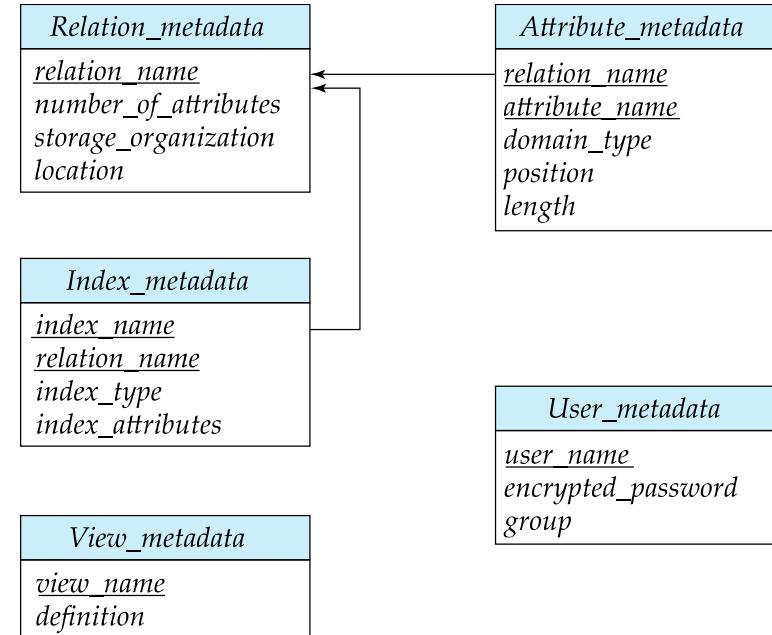
The **Data dictionary** (also called **system catalog**) stores **metadata**; that is, data about data, such as

- 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)

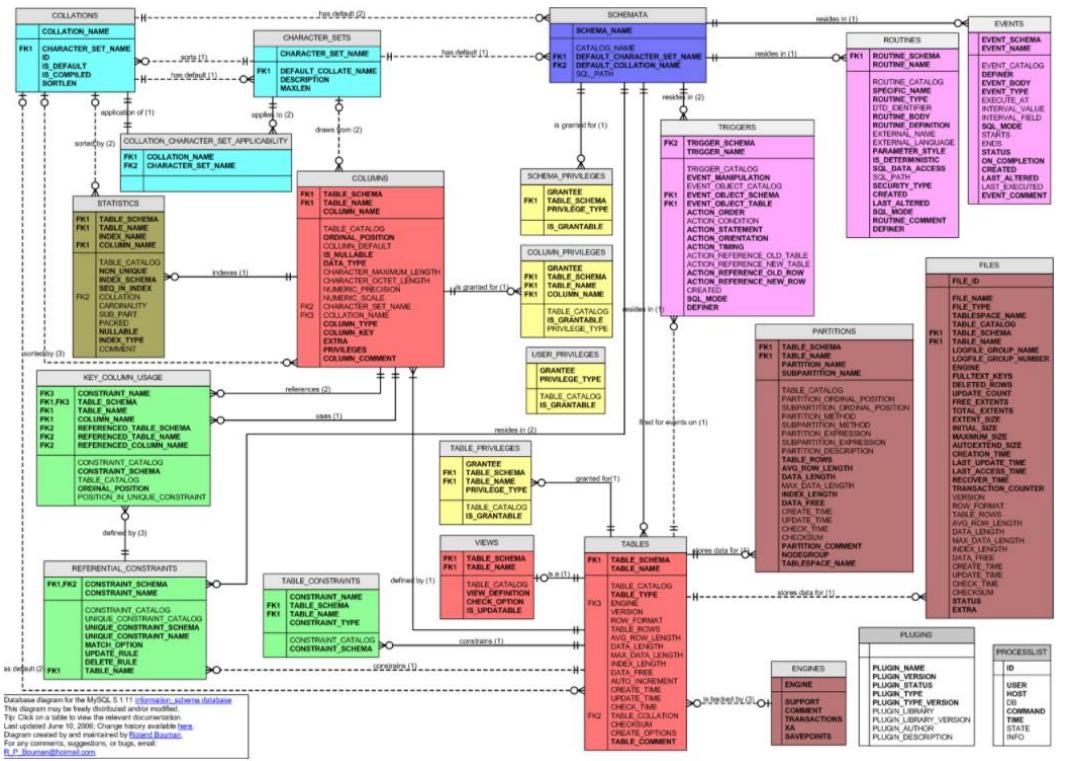


# Relational Representation of System Metadata

- 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 involve 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

# ER Modeling

# Degree Cardinality



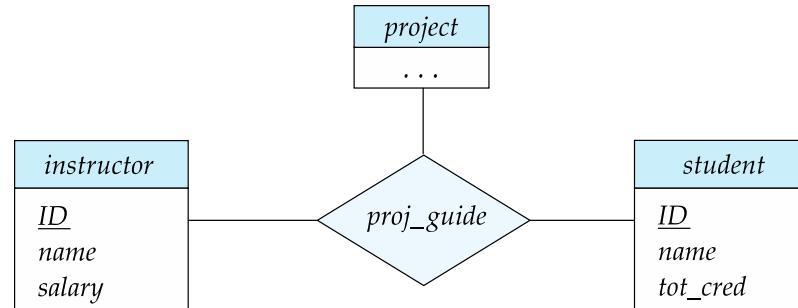
# Degree of a Relationship Set

- Binary relationship
  - involve two entity sets (or degree two).
  - most relationship sets in a database system are binary.
- Relationships between more than two entity sets are rare. Most relationships are binary. (More on this later.)
  - Example: *students* work on research *projects* under the guidance of an *instructor*.
  - relationship *proj\_guide* is a ternary relationship between *instructor*, *student*, and *project*



# Non-binary Relationship Sets

- Most relationship sets are binary
- There are occasions when it is more convenient to represent relationships as non-binary.
- E-R Diagram with a Ternary Relationship





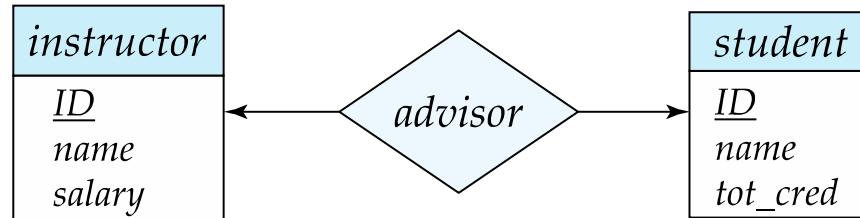
# Mapping Cardinality Constraints

- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
  - One to one
  - One to many
  - Many to one
  - Many to many



# Representing Cardinality Constraints in ER Diagram

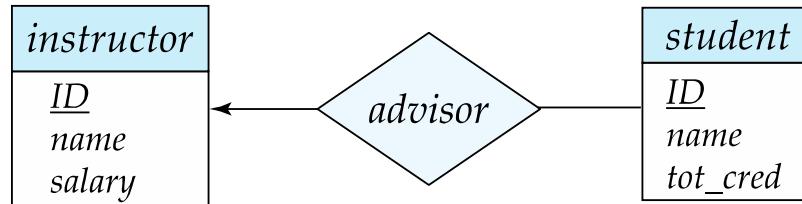
- We express cardinality constraints by drawing either a directed line ( $\rightarrow$ ), signifying “one,” or an undirected line ( $-$ ), signifying “many,” between the relationship set and the entity set.
- One-to-one relationship between an *instructor* and a *student* :
  - A student is associated with at most one *instructor* via the relationship *advisor*
  - A *student* is associated with at most one *department* via *stud\_dept*





# One-to-Many Relationship

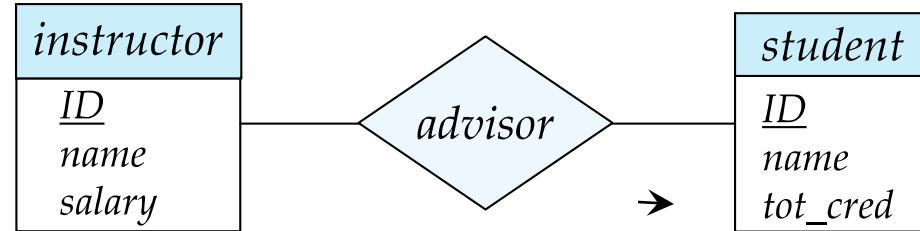
- one-to-many relationship between an *instructor* and a *student*
  - an instructor is associated with several (including 0) students via *advisor*
  - a student is associated with at most one instructor via advisor,





# Many-to-One Relationships

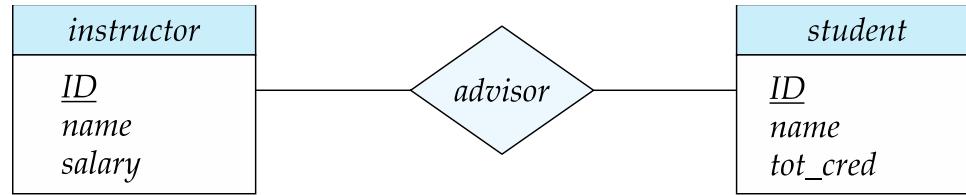
- In a many-to-one relationship between an *instructor* and a *student*,
  - an *instructor* is associated with at most one *student* via *advisor*,
  - and a *student* is associated with several (including 0) *instructors* via *advisor*





# Many-to-Many Relationship

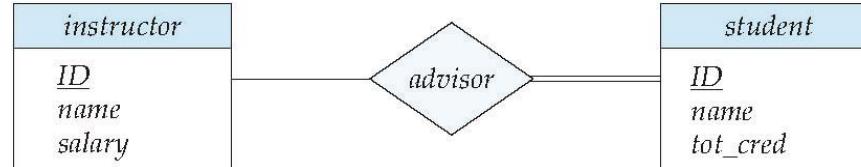
- An instructor is associated with several (possibly 0) students via *advisor*
- A student is associated with several (possibly 0) instructors via *advisor*





# Total and Partial Participation

- **Total participation** (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set



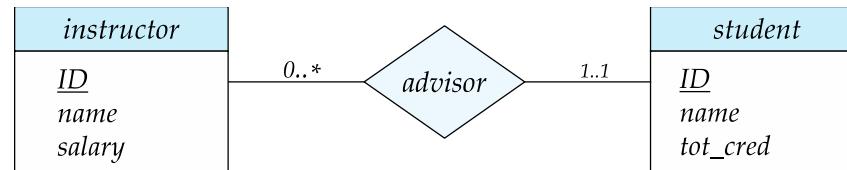
participation of *student* in *advisor* relation is total

- every *student* must have an associated instructor
- **Partial participation:** some entities may not participate in any relationship in the relationship set
  - Example: participation of *instructor* in *advisor* is partial



# Notation for Expressing More Complex Constraints

- A line may have an associated minimum and maximum cardinality, shown in the form  $l..h$ , where  $l$  is the minimum and  $h$  the maximum cardinality
  - A minimum value of 1 indicates total participation.
  - A maximum value of 1 indicates that the entity participates in at most one relationship
  - A maximum value of \* indicates no limit.
- Example



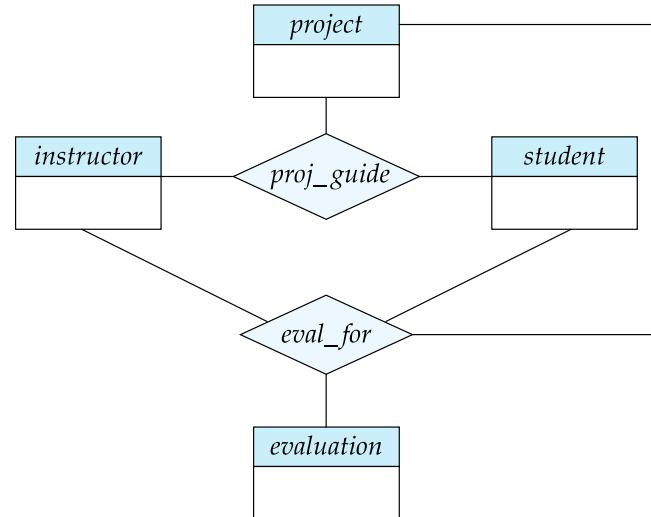
- Instructor can advise 0 or more students. A student must have 1 advisor; cannot have multiple advisors

# 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





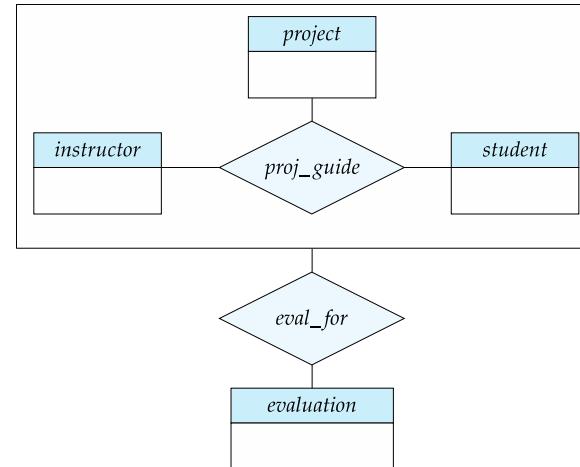
# Aggregation (Cont.)

- 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



# 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:

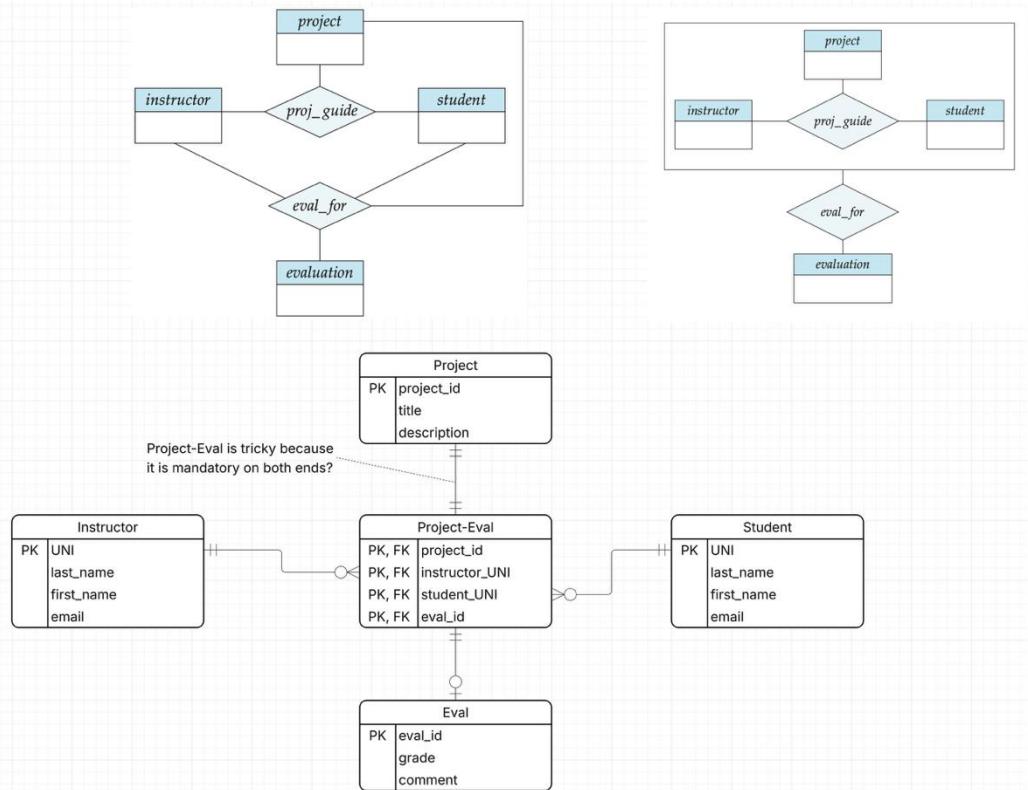
<https://www.geeksforgeeks.org/aggregate-data-model-in-nosql/>



# 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:  
 $\text{eval\_for}(\text{s\_ID}, \text{project\_id}, \text{i\_ID}, \text{evaluation\_id})$
  - The schema *proj\_guide* is redundant.

# Let's Do a More Example in Crow's Foot



If I got that right, it is a miracle.

Normally, you cannot be sure until you implement and test it.

# Relational Model

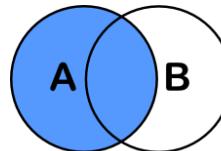
# What are all those other Symbols?

- $\tau$  order by
- $\gamma$  group by
- $\neg$  negation
- $\div$  set division
- $\bowtie$  natural join, theta-join
- $\bowtie_l$  left outer join
- $\bowtie_r$  right outer join
- $\bowtie_f$  full outer join
- $\bowtie_{lsj}$  left semi join
- $\bowtie_{rsj}$  right semi join
- $\triangleright$  anti-join

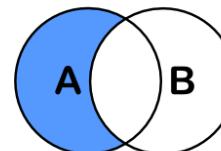
- Some of the operators are useful and “common,” but not always considered part of the core algebra.
- Some of these are pretty obscure
  - Division
  - Anti-Join
  - Left semi-join
  - Right semi-join
- Most SQL engines do not support them.
  - You can implement them using combinations of JOIN, SELECT, 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 operators because they may be common internships/job interview questions.

# Thinking about JOINS

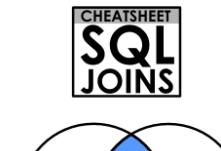
- Some terms:
  - Natural Join
    - Equality of A and B columns
    - With the same name.
  - Equijoin
    - Explicitly specify columns that must have the same value.
    - $A.x=B.z \text{ AND } A.q=B.w$
  - Theta Join: Arbitrary predicate.
- Inner Join
  - JOIN “matches” rows in A and B.
  - Result 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 ( $\bowtie$ ,  $\bowtie\bowtie$ )



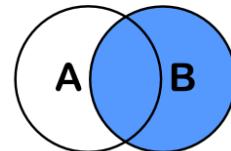
```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
```



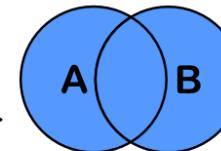
```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
WHERE B.key IS NULL
```



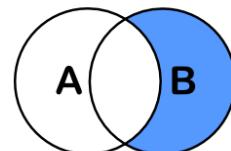
```
SELECT <auswahl>
FROM tabelleA A
INNER JOIN tabelleB B
ON A.key = B.key
```



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



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



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



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

# What are all those other Symbols?

- $\tau$  order by
- $\gamma$  group by
- $\neg$  negation
- $\div$  set division
- $\bowtie$  natural join, theta-join
- $\bowtie_l$  left outer join
- $\bowtie_r$  right outer join
- $\bowtie_f$  full outer join
- $\bowtie_{lsj}$  left semi join
- $\bowtie_{rsj}$  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 operators because 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  $\triangleright$  ID=i\_id advisor

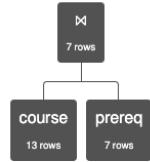
instructor.ID	instructor.name	instructor.dept_name	instructor.salary
12121	'Wu'	'Finance'	90000
15151	'Mozart'	'Music'	40000
32343	'El Said'	'History'	60000
33456	'Gold'	'Physics'	87000
58583	'Califieri'	'History'	62000
83821	'Brandt'	'Comp. Sci.'	92000

$\sigma i\_id=null$  (instructor  $\bowtie$  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  $\bowtie$  prereq

Execution time: 1 ms



course  $\bowtie$  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.

$$\tau \text{total\_seats } (\gamma \text{ building; sum(capacity)} \rightarrow \text{total\_seats} \text{ (classroom)})$$

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

# Division

## Division [\[edit\]](#)

The division ( $\div$ ) 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 [\[edit\]](#)

### Relax Example

$X = \pi_{course\_id} \leftarrow course\_id (\sigma_{dept\_name='History'}(course))$

$Y = \pi_{ID} \leftarrow ID, course\_id \leftarrow course\_id (student \bowtie takes)$

$Y \div X$

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:

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

where  $\{a_1, \dots, a_n\}$  is the set of attribute names unique to  $R$  and  $t[a_1, \dots, a_n]$  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.

# SQL

# *Some Types and Functions*

# Functions

## MySQL CHEAT SHEET: STRING FUNCTIONS

by [sqlbackupandftp.com](http://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 ('Ø') = 2`  
`LENGTH ('A') = 1`  
`OCTET_LENGTH ('Ø') = 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 ('ð') = 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' USING utf8)) = 'utf8'`

Decode to / from a base-64 string

`DECODE ('abc') = 'YnDj'`  
`FROM_BASE64 ('YnDj') = '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', ' ', 'Sql') = 'MySQL'`  
`CONCAT ('My', NULL, 'Sql') = 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, ':') = 'S:ql'`  
`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'`

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

`RPAD ('Sql', 2, ':') = 'Sql'`  
`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 1) = 'q1'`  
`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','Y','N',4) = 'Y,N,Y,N'`  
`EXPORT_SET (6, '1','0','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') = ''`

### 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 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(13.8); -- 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 `NULLS`. 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`.

`NULIF(x, y)`

To save yourself from `division by 0` errors:

```
SELECT
last_month,
this_month,
this_month * 100.0
/ NULIF(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 `NULIF(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 a 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 `NULIF()` function to replace 0 with a `NULL`, which will result in a `NULL` for the whole expression:  
`count / NULIF(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](https://LearnSQL.com), and check out our other SQL courses.

LearnSQL.com is owned by Vertabelo SA  
vertabelo.com | CC-BY-NC-ND Vertabelo SA

# Functions

## Standard SQL Functions Cheat Sheet

### 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-NUL 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
------	------

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

#### 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: 123 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 (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 INTERVALS 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 INTERVAL 'x' year y month'	
INTERVAL 'x-y' DAY TO 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` type.

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

- There 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!

# *Indexes*



# 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

# *Functions, Procedures and Trigger Concepts*



# Functions and Procedures

- Functions and procedures allow “business logic” to be stored in the database and executed from SQL statements.
- These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++.
- The syntax we present here is defined by the SQL standard.
  - Most databases implement nonstandard versions of this syntax.

## Note:

- The programming language, runtime and tools for functions, procedures and triggers are not easy to use.
- My view is that calling external functions is an anti-pattern (bad idea).
  - External code degrades the reliability, security and performance of the database.
  - Databases are often mission critical and the heart of environments.



# Language Constructs for Procedures & Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
  - Warning: most database systems implement their own variant of the standard syntax below.
- Compound statement: **begin** ... **end**, {}
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements
- While and repeat statements:
  - **while** boolean expression **do**  
sequence of statements ;  
**end while**
  - **repeat**  
sequence of statements ;  
until boolean expression  
**end repeat**



## (Core) Language Constructs (Cont.)

- **For** loop
  - Permits iteration over all results of a query
- Example: Find the budget of all departments

```
declare n integer default 0;
for r as
    select budget from department
        where dept_name = 'Music'
do
    set n = n + r.budget
end for
```

### Note:

- There are various other looping constructs.



# (Core) Language Constructs – if-then-else

- Conditional statements (**if-then-else**)

```
if boolean expression
    then statement or compound statement
    elseif boolean expression
        then statement or compound statement
    else statement or compound statement
end if
```

## Note:

- We will not spend a lot of time writing functions, procedures, or triggers.
- The language and development environment are not easy to use.

# *Functions*





# Declaring SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
    returns integer
begin
    declare d_count integer;
    select count (*) into d_count
        from instructor
        where instructor.dept_name = dept_name
    return d_count;
end
```

- The function *dept\_count* can be used to find the department names and budget of all departments with more than 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```



# Table Functions

- The SQL standard supports functions that can return tables as results; such functions are called **table functions**
- Example: Return all instructors in a given department

```
create function instructor_of (dept_name char(20))  
    returns table (  
        ID varchar(5),  
        name varchar(20),  
        dept_name varchar(20),  
        salary numeric(8,2))  
  
return table  
(select ID, name, dept_name, salary  
from instructor  
where instructor.dept_name = instructor_of.dept_name)
```

- Usage

```
select *  
from table (instructor_of ('Music'))
```

# *Procedures*



# SQL Procedures

- The *dept\_count* function could instead be written as procedure:

```
create procedure dept_count_proc (in dept_name varchar(20),
                                   out d_count integer)
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_count_proc.dept_name
end
```

- The keywords **in** and **out** are parameters that are expected to have values assigned to them and parameters whose values are set in the procedure in order to return results.
- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

```
declare d_count integer;
call dept_count_proc('Physics', d_count);
```



## SQL Procedures (Cont.)

- Procedures and functions can be invoked also from dynamic SQL
- SQL allows more than one procedure of the same name so long as the number of arguments of the procedures with the same name is different.
- The name, along with the number of arguments, is used to identify the procedure.

# *Triggers*



# Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  - Syntax illustrated here may not work exactly on your database system; check the system manuals

X on T

before insert

before update

before delete

after insert

after update

after delete



# Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - For example, **after update of takes on grade**
- Values of attributes before and after an update can be referenced
  - **referencing old row as** : for deletes and updates
  - **referencing new row as** : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
    when (nrow.grade = ' ')
begin atomic
    set nrow.grade = null;
end;
```



# Trigger to Maintain credits\_earned value

- **create trigger *credits\_earned* after update of *takes* on (*grade*) referencing new row as *nrow* referencing old row as *orow* for each row when *nrow.grade* <> 'F' and *nrow.grade* is not null and (*orow.grade* = 'F' or *orow.grade* is null) begin atomic update *student* set *tot\_cred*= *tot\_cred* + (select *credits* from *course* where *course.course\_id*= *nrow.course\_id*) where *student.id* = *nrow.id*; end;**



# Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called **transition tables**) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows



# When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - Maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger



# When Not To Use Triggers (Cont.)

- Risk of unintended execution of triggers, for example, when
  - Loading data from a backup copy
  - Replicating updates at a remote site
  - Trigger execution can be disabled before such actions.
- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
  - Cascading execution

# *Summary*



# Comparison

## comparing triggers, functions, and procedures

	triggers	functions	stored procedures
change data	yes	no	yes
return value	never	always	sometimes
how they are called	reaction	in a statement	exec

# Comparison – Some Details

A *trigger* has capabilities like a procedure, except ...

- You do not call it. The DB engine calls it before or after an INSERT, UPDATE, DELETE.
- The inputs are the list of incoming new, modified rows.
- The outputs are the modified versions of the new or modified rows.

Sr.No.	User Defined Function	Stored Procedure
1	Function must return a value.	Stored Procedure may or not return values.
2	Will allow only Select statements, it will not allow us to use DML statements.	Can have select statements as well as DML statements such as insert, update, delete and so on
3	It will allow only input parameters, doesn't support output parameters.	It can have both input and output parameters.
4	It will not allow us to use try-catch blocks.	For exception handling we can use try catch blocks.
5	Transactions are not allowed within functions.	Can use transactions within Stored Procedures.
6	We can use only table variables, it will not allow using temporary tables.	Can use both table variables as well as temporary table in it.
7	Stored Procedures can't be called from a function.	Stored Procedures can call functions.
8	Functions can be called from a select statement.	Procedures can't be called from Select/Where/Having and so on statements. Execute/Exec statement can be used to call/execute Stored Procedure.
9	A UDF can be used in join clause as a result set.	Procedures can't be used in Join clause

# *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

## 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.

## 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.

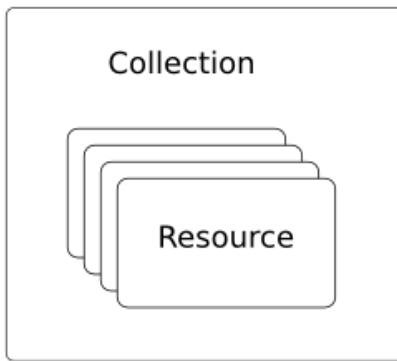
## Creating RESTful Webservice

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

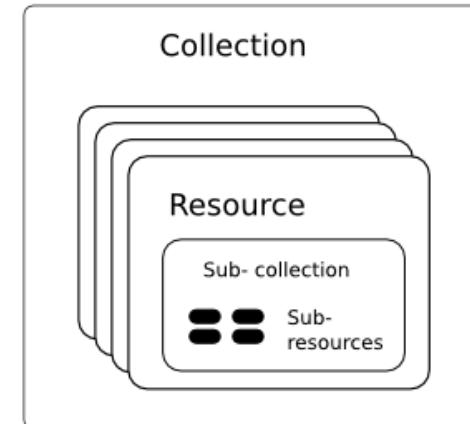
## Resource Model



A Collection with  
Resources

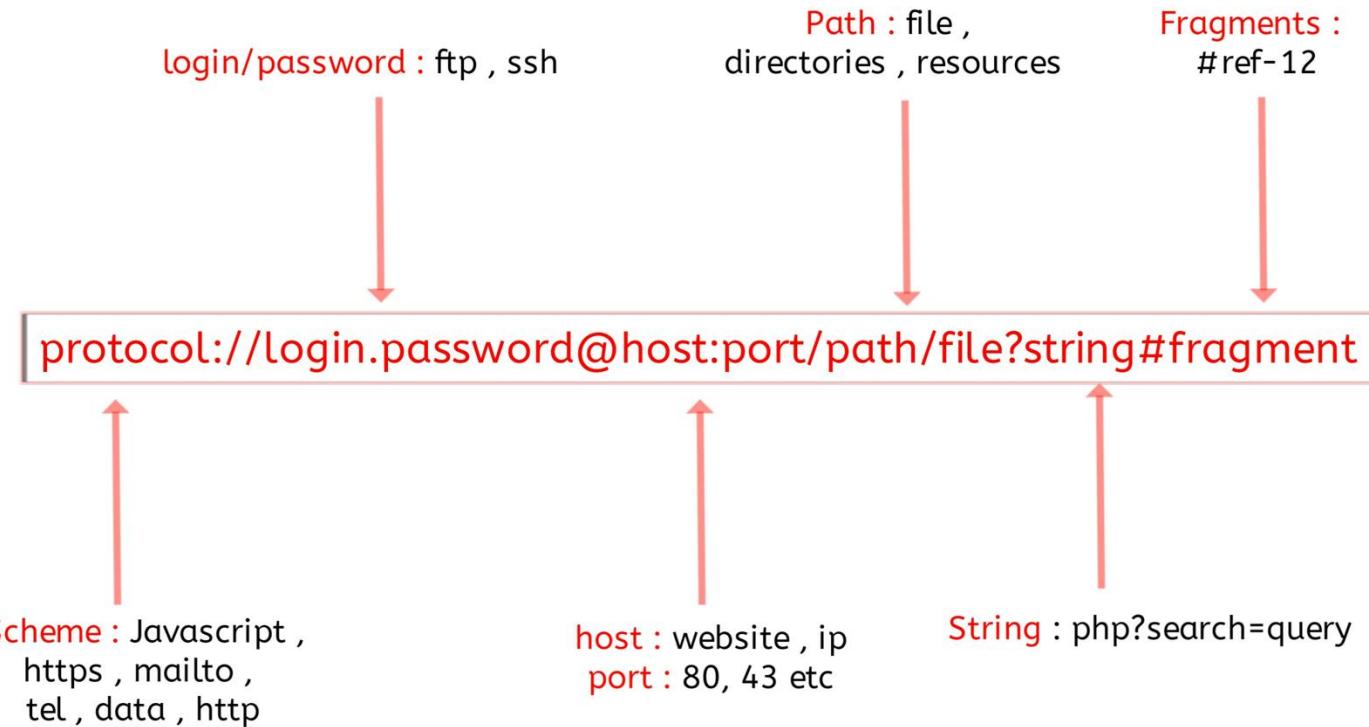


A Singleton  
Resource



Sub-collections and  
Sub-resources

# URLs



# 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

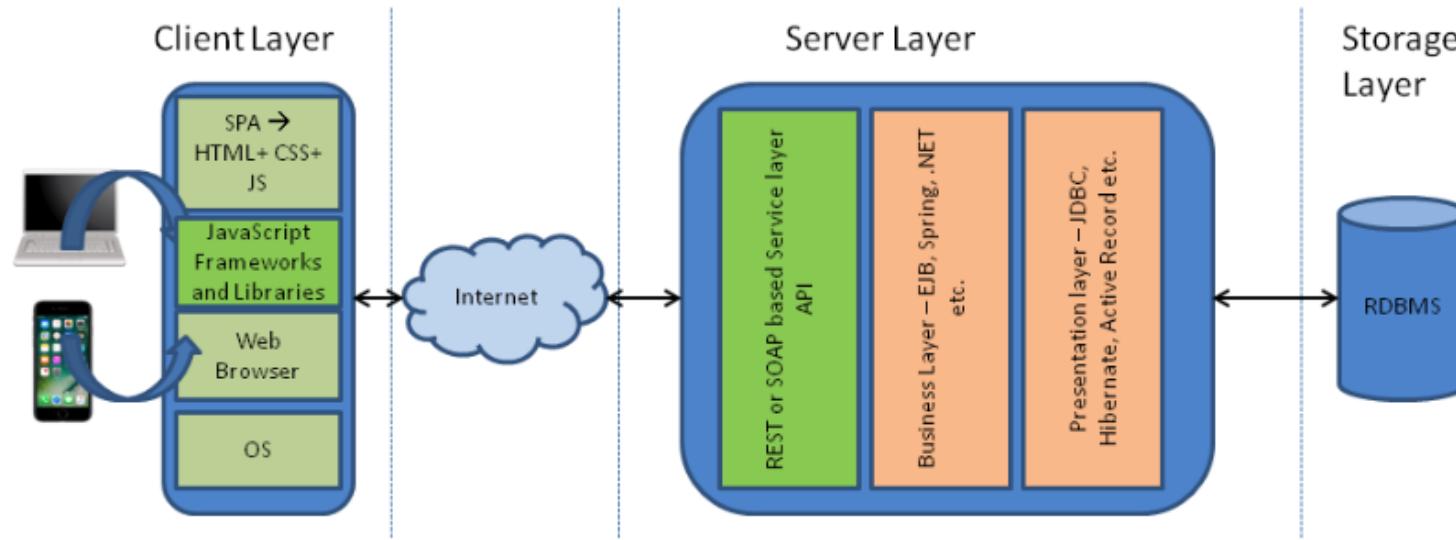


Diagram 2: The moving of the Web Layer from the Server to the Client