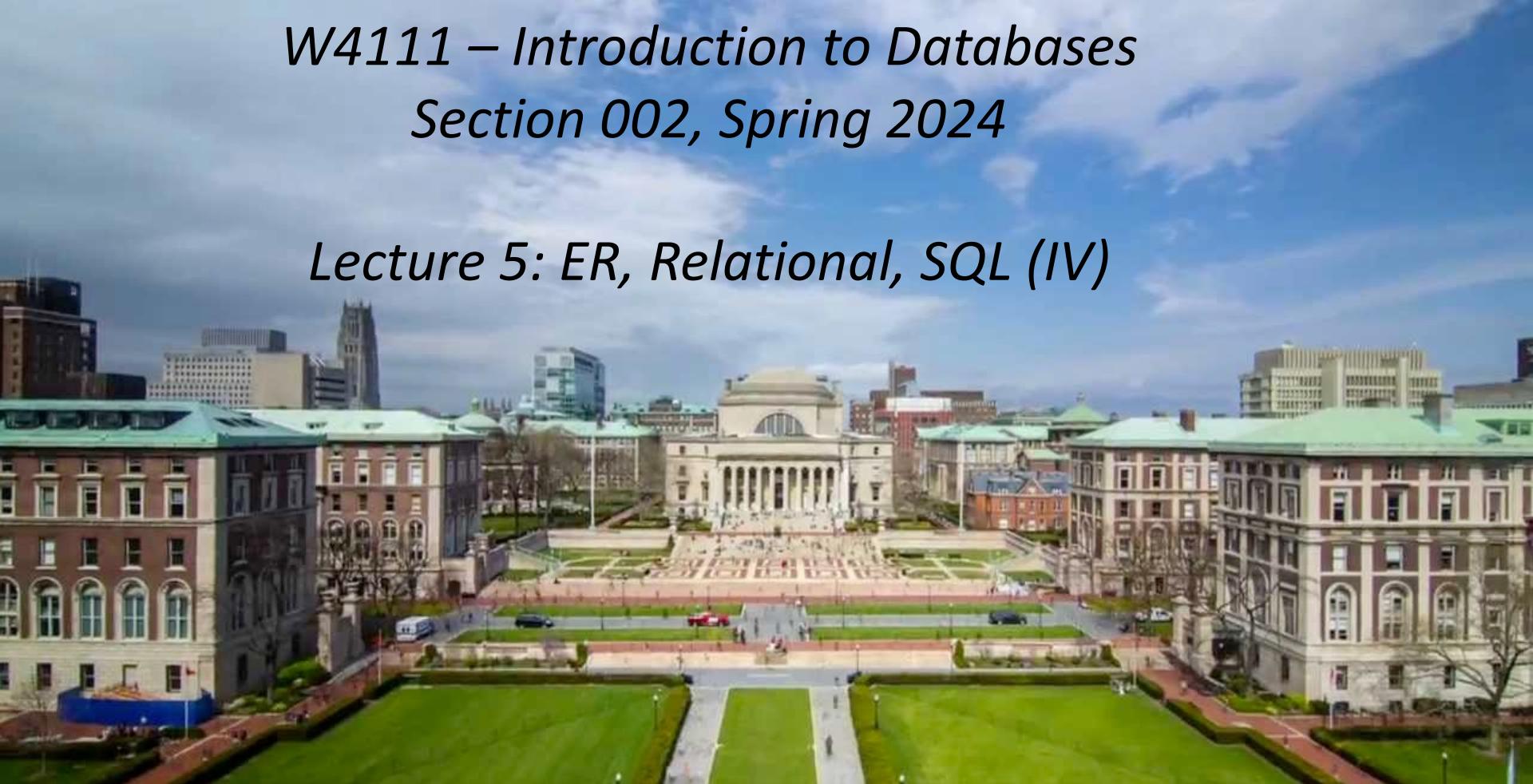


*W4111 – Introduction to Databases
Section 002, Spring 2024*

Lecture 5: ER, Relational, SQL (IV)



W4111 – Introduction to Databases

Section 002, Spring 2024

Lecture 5: ER, Relational, SQL (IV)

We will start in a couple of minutes.

Contents

Contents

- Some Observations
- ER (Diagram) Modeling – More Complex Scenarios
- The Relational Model and Algebra Continued (Chapter 3) – Just Kidding
- SQL Continued (Chapter 3, 4, 5)
- Project Examples:
 - Web Applications and REST
 - Data Engineering and Visualization

Some Observations

Database design, Entity-Relationship Model Some Complex Examples



Weak Entity Sets

- Consider a *section* entity, which is uniquely identified by a *course_id*, *semester*, *year*, and *sec_id*.
- Clearly, section entities are related to course entities. Suppose we create a relationship set *sec_course* between entity sets *section* and *course*.
- Note that the information in *sec_course* is redundant, since *section* already has an attribute *course_id*, which identifies the course with which the section is related.
- One option to deal with this redundancy is to get rid of the relationship *sec_course*; however, by doing so the relationship between *section* and *course* becomes implicit in an attribute, which is not desirable.



Weak Entity Sets (Cont.)

- An alternative way to deal with this redundancy is to not store the attribute *course_id* in the *section* entity and to only store the remaining attributes *section_id*, *year*, and *semester*.
 - However, the entity set *section* then does not have enough attributes to identify a particular *section* entity uniquely
- To deal with this problem, we treat the relationship *sec_course* as a special relationship that provides extra information, in this case, the *course_id*, required to identify *section* entities uniquely.
- A **weak entity set** is one whose existence is dependent on another entity, called its **identifying entity**
- Instead of associating a primary key with a weak entity, we use the identifying entity, along with extra attributes called **discriminator** to uniquely identify a weak entity.



Weak Entity Sets (Cont.)

- An entity set that is not a weak entity set is termed a **strong entity set**.
- Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set.
- The identifying entity set is said to **own** the weak entity set that it identifies.
- The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.
- Note that the relational schema we eventually create from the entity set *section* does have the attribute *course_id*, for reasons that will become clear later, even though we have dropped the attribute *course_id* from the entity set *section*.

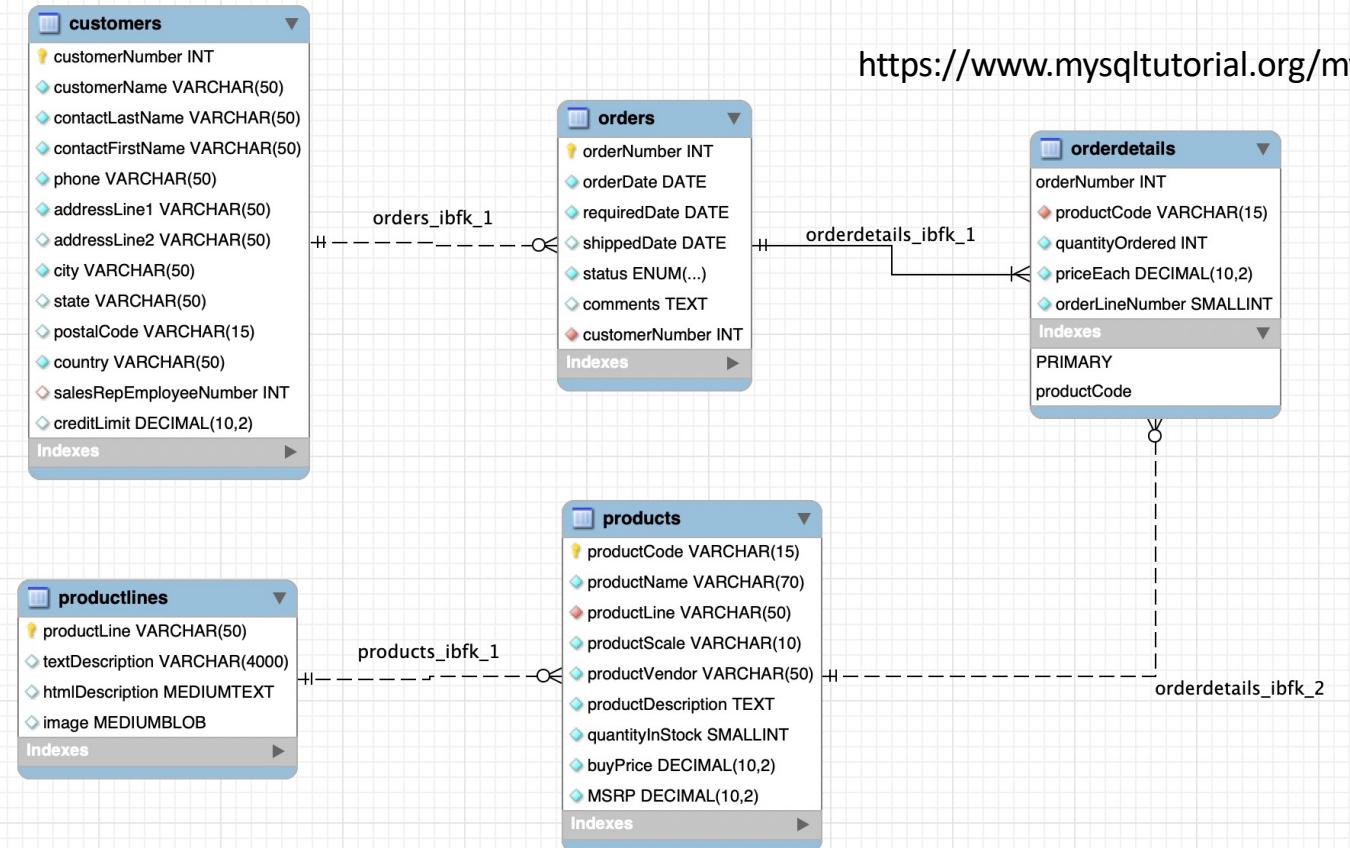


Expressing Weak Entity Sets

- In E-R diagrams, a weak entity set is depicted via a double rectangle.
- We underline the discriminator of a weak entity set with a dashed line.
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond.
- Primary key for *section* – (*course_id*, *sec_id*, *semester*, *year*)



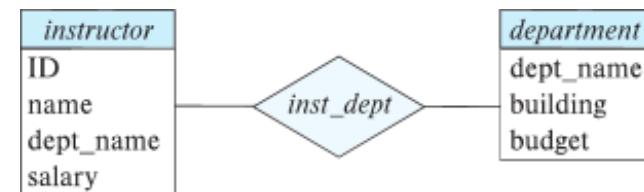
An Example – Classic Models



<https://www.mysqltutorial.org/mysql-sample-database.aspx/>

Redundant Attributes

- Suppose we have entity sets:
 - *instructor*, with attributes: *ID, name, dept_name, salary*
 - *department*, with attributes: *dept_name, building, budget*
- We model the fact that each instructor has an associated department using a relationship set *inst_dept*
- The attribute *dept_name* in *instructor* replicates information present in the relationship and is therefore redundant
 - and needs to be removed.
- BUT: when converting back to tables, in some cases the attribute gets reintroduced, as we will see later.





Design Alternatives

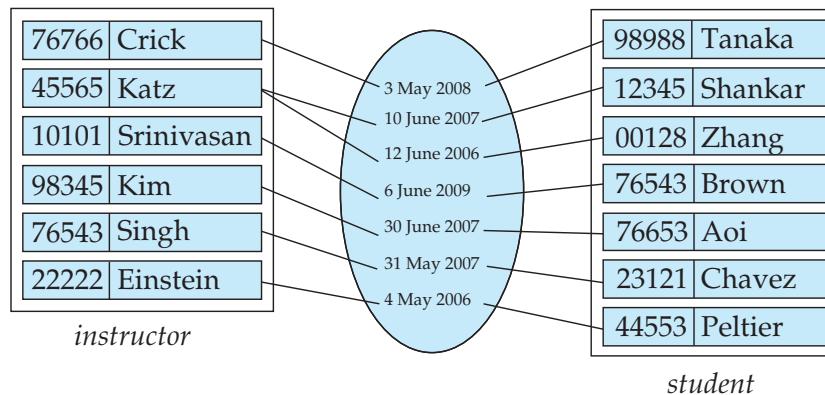
- In designing a database schema, we must ensure that we avoid two major pitfalls:
 - Redundancy: a bad design may result in repeat information.
 - **Redundant representation of information may lead to data inconsistency among the various copies of information**
 - Incompleteness: a bad design may make certain aspects of the enterprise difficult or impossible to model.
- Avoiding bad designs is not enough. There may be a large number of good designs from which we must choose.

**Emphasis
Added**



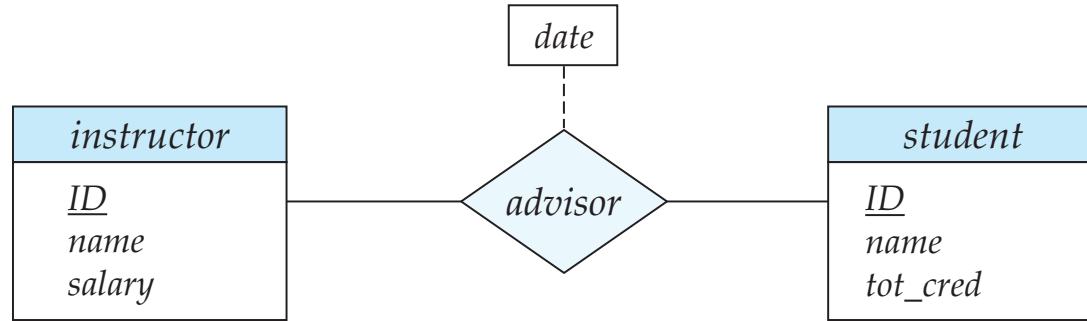
Relationship Sets (Cont.)

- An attribute can also be associated with a relationship set.
- For instance, the *advisor* relationship set between entity sets *instructor* and *student* may have the attribute *date* which tracks when the student started being associated with the advisor





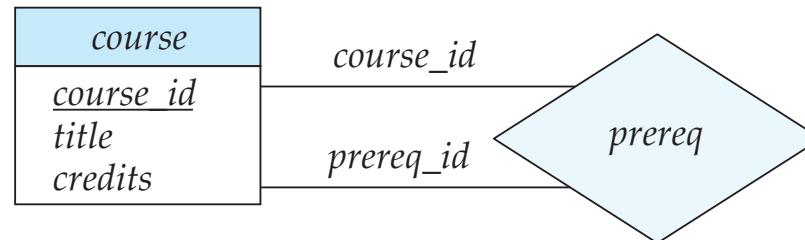
Relationship Sets with Attributes





Roles

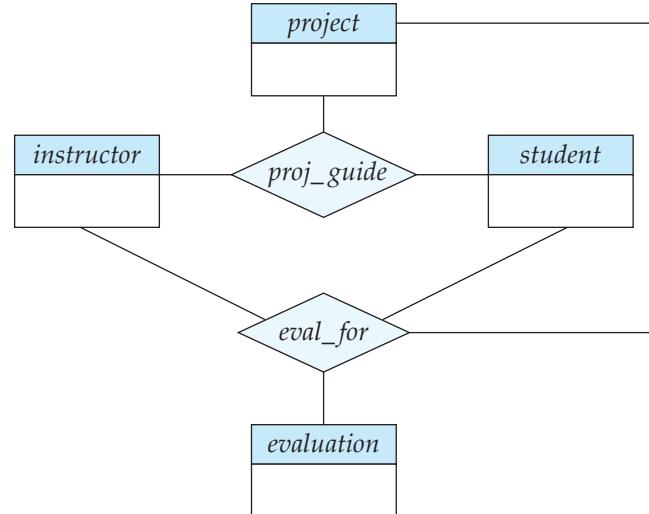
- Entity sets of a relationship need not be distinct
 - Each occurrence of an entity set plays a “role” in the relationship
- The labels “*course_id*” and “*prereq_id*” are called **roles**.





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





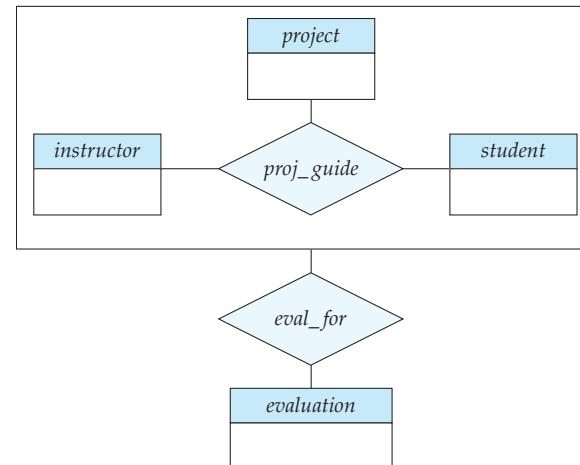
Aggregation (Cont.)

- 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:
$$\text{eval_for} (s_ID, project_id, i_ID, evaluation_id)$$
 - The schema *proj_guide* is redundant.



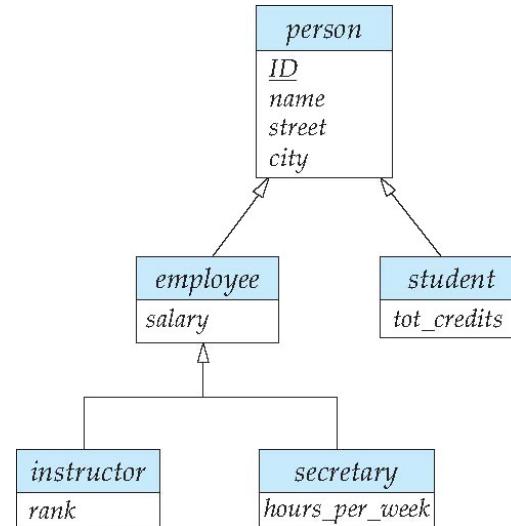
Specialization

- 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.



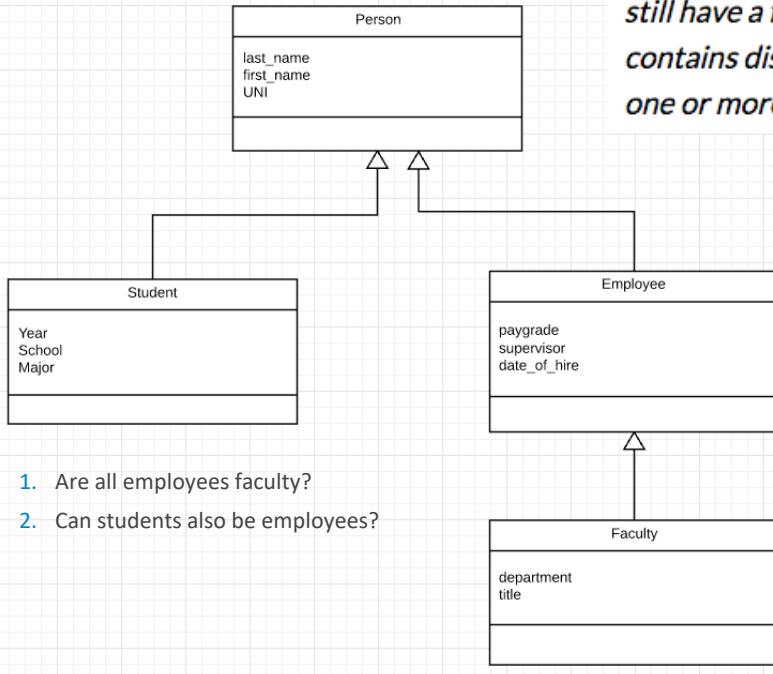
Specialization Example

- **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

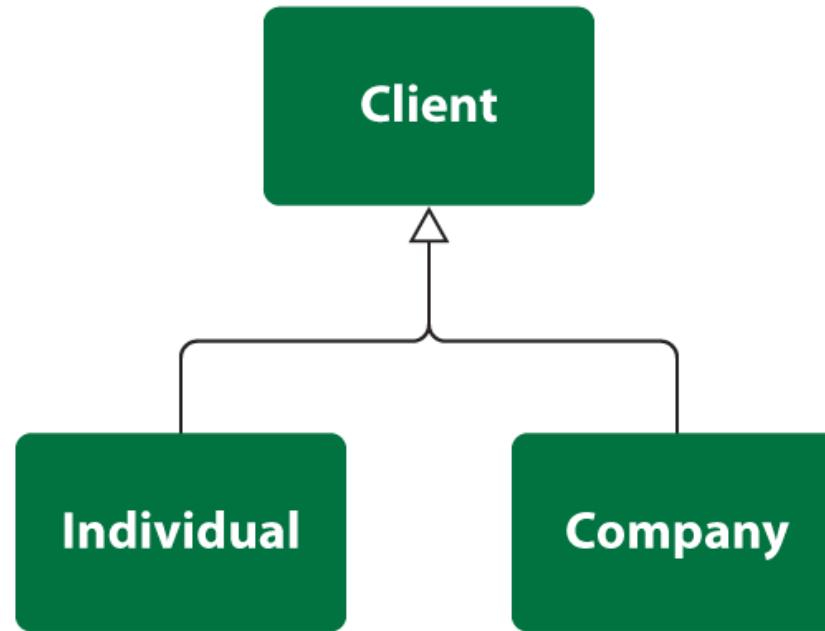
- In an **incomplete** specialization only some instances of the parent class are specialized (have unique attributes). Other instances of the parent class have only the common attributes.
- In a **complete** specialization, every instance of the parent class has one or more unique attributes that are 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.

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).

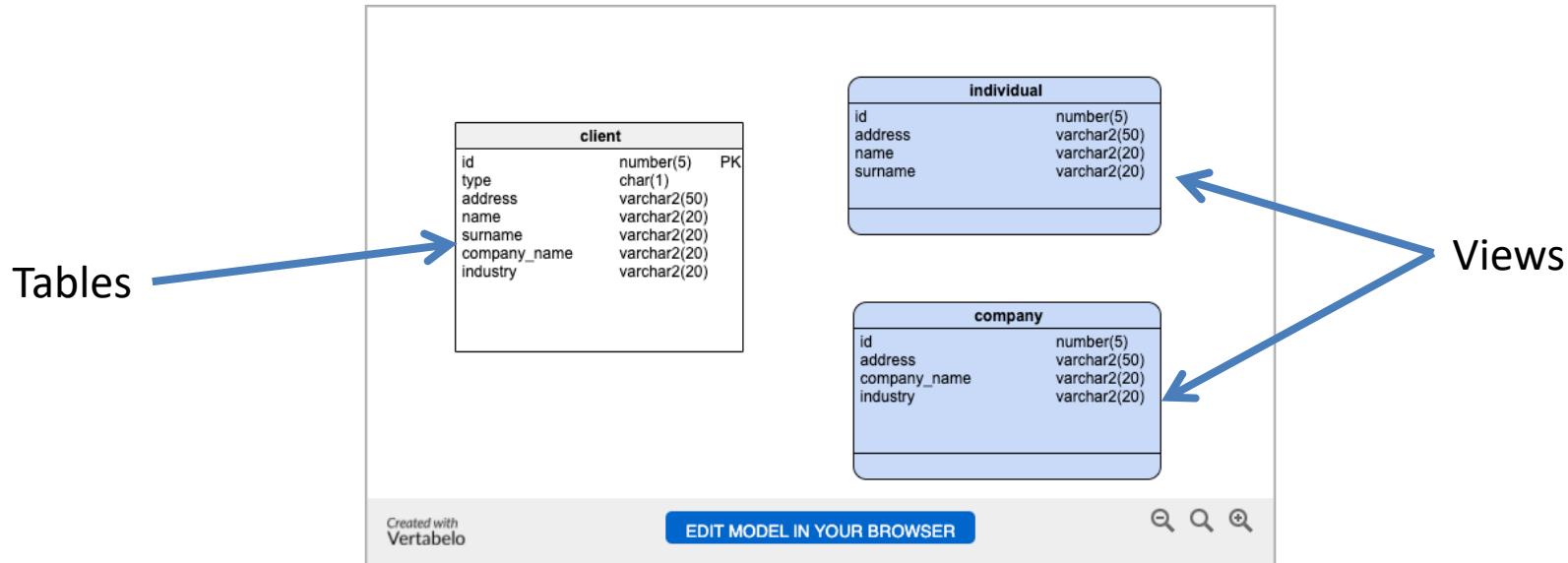


One Table

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`:

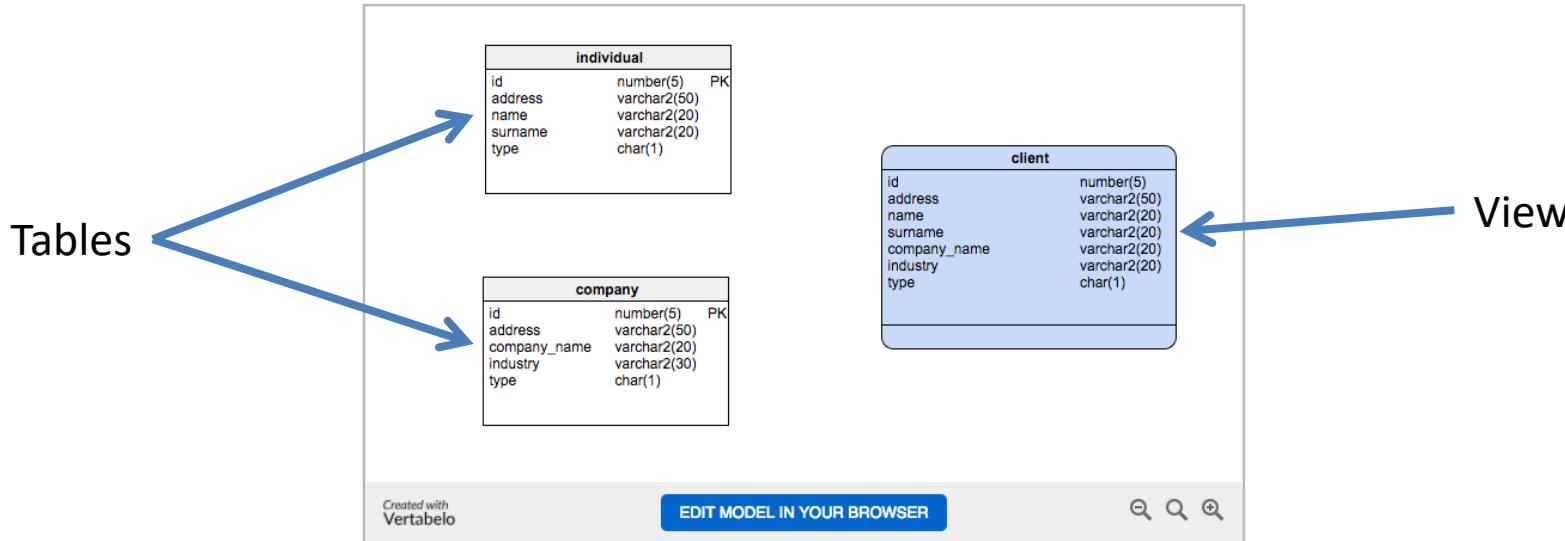


Two Table

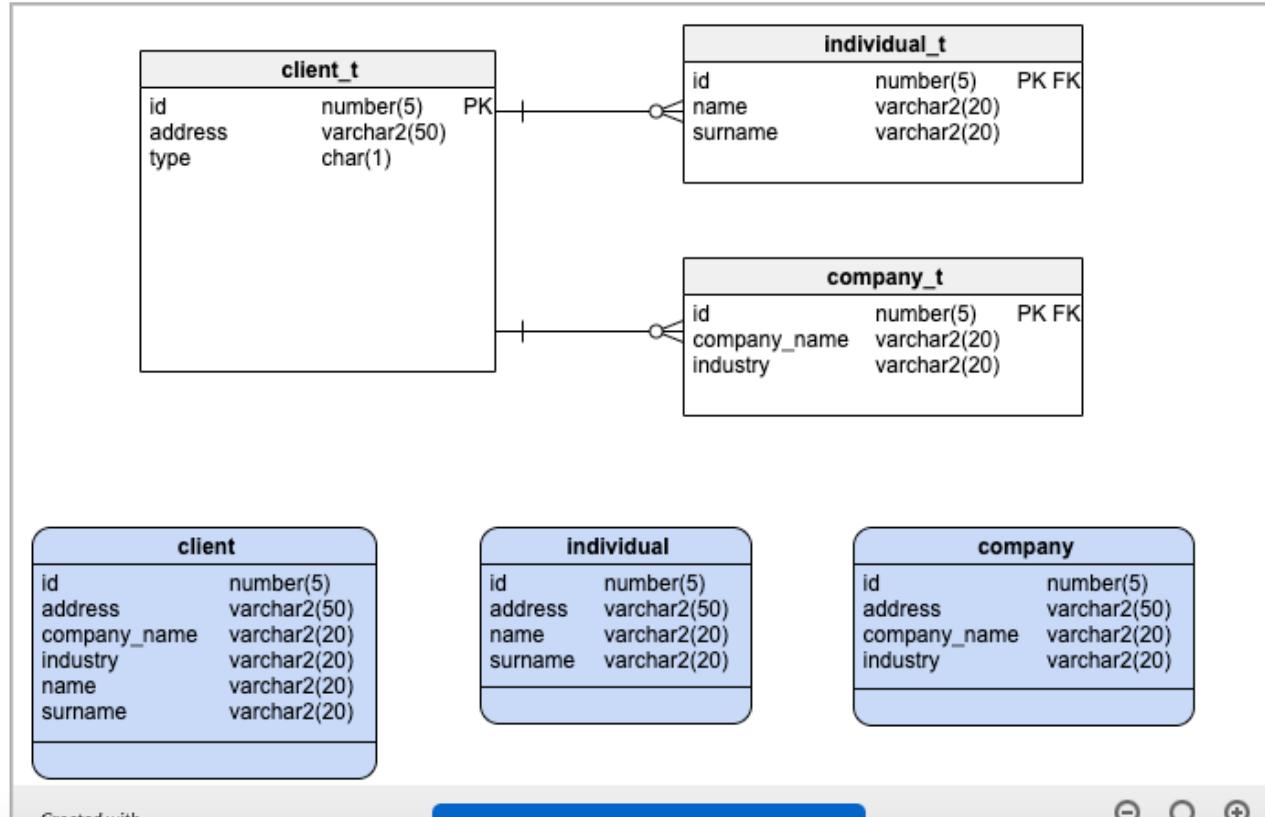
Two-table implementation

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

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



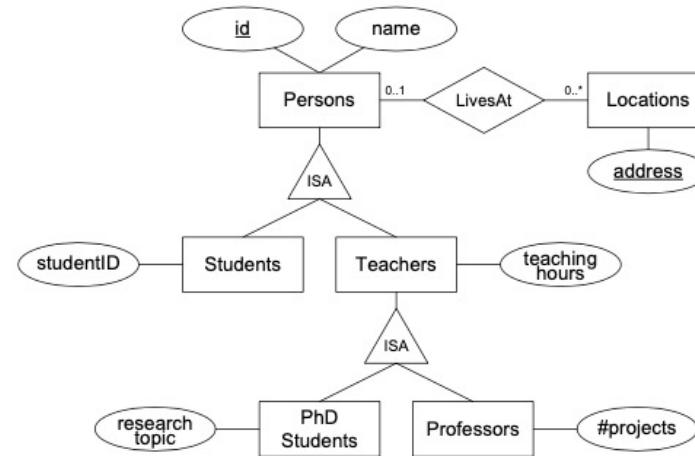
Three Table



ISA Relationship



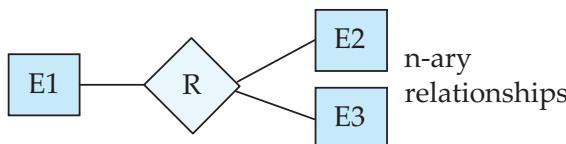
ISA Relationship





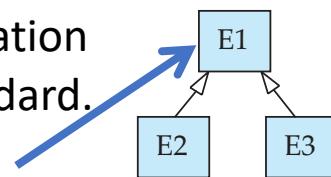
ER vs. UML Class Diagrams

ER Diagram Notation

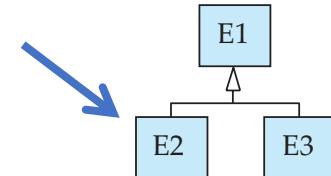


n-ary
relationships

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

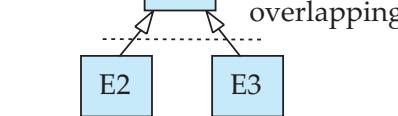
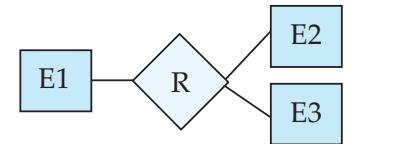


overlapping
generalization

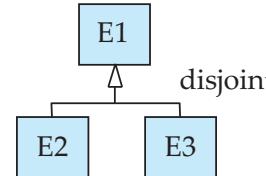


disjoint
generalization

Equivalent in UML



overlapping



disjoint

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

Relation Model and Algebra

Relation Model and Algebra

Just kidding!
We are basically done with relational algebra!

SQL

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 rules 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.

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 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

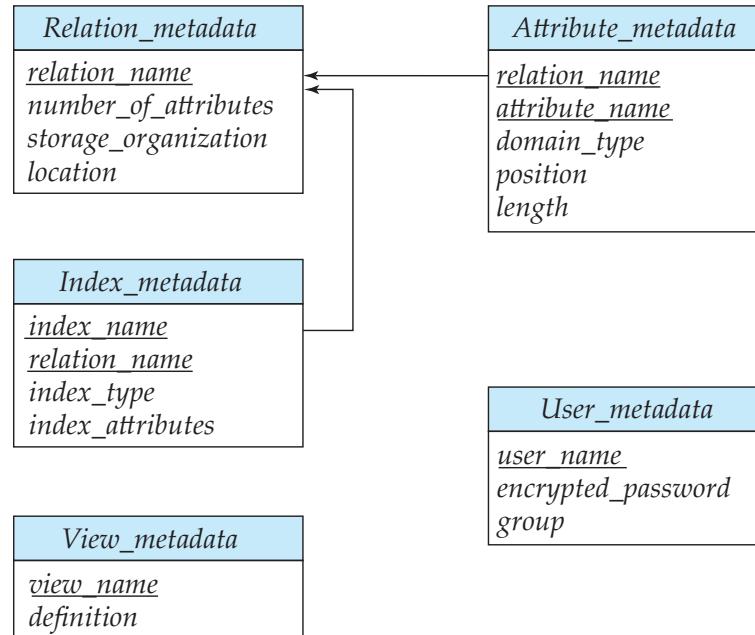
Data Dictionary Storage

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

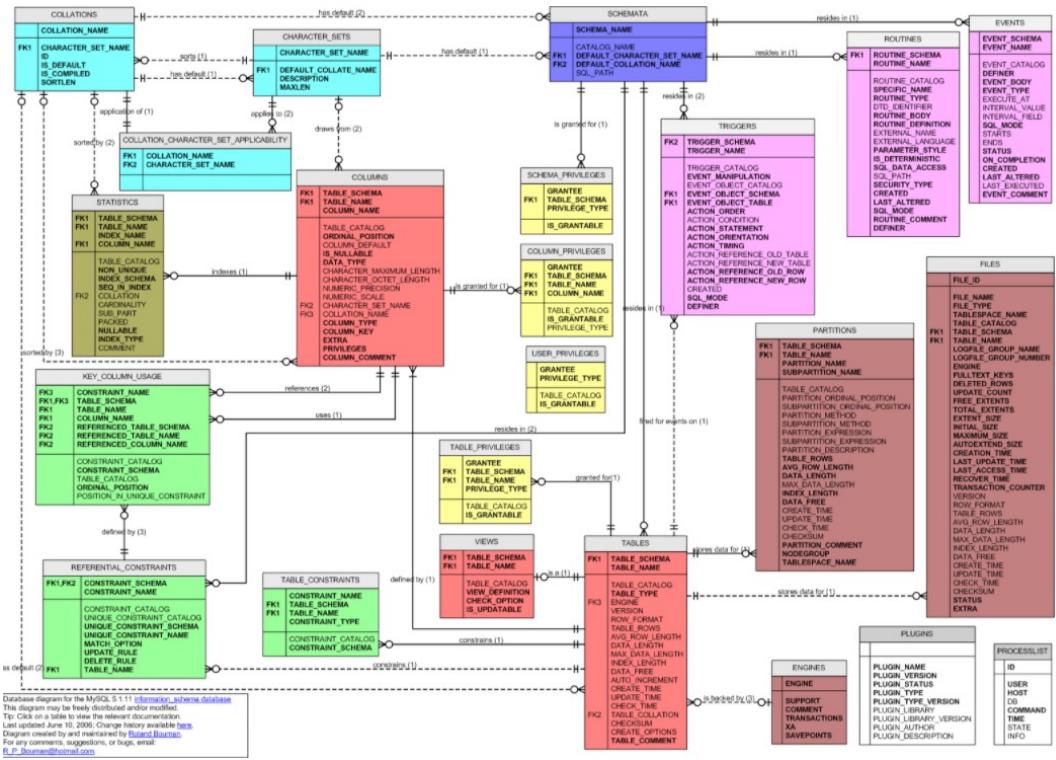
- Information about relations
 - names of relations
 - names, types and lengths of attributes of each relation
 - names and definitions of views
 - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
 - number of tuples in each relation
- Physical 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'
- 'VIEWS'
- 'CREATE and ALTER statements modify the data.'
- 'DBMS reads information: Parsing, Optimizer, etc.'

Metadata

Notebook examples.

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 ('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'65' USING utf8)) = 'utf8'`

Decode to / from a base-64 string

`DECODE ('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', ' ', '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., 4) = 12,332.1000`
`FORMAT ('12332.,2) = 12332.2`
`FORMAT ('12332.,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, ':') = 'S:q'`
`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, ':') = 'S:q'`
`RPAD ('Sql', 4, ':') = 'S:ql'`
`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:1'`
`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') = ''`

Standard SQL Functions Cheat Sheet

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(13.1); -- 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 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 'C'
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 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, 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;
```

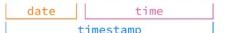
Common mistake: COUNT(*) and LEFT JOIN

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



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' year y month'
INTERVAL 'x-y' DAY TO INTERVAL 'x' day y 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

- 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!

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
 - Insert 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.

# *Integrity Constraints*



# Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than \$10,000.00
  - A salary of a bank employee must be at least \$4.00 an hour
  - A customer must have a (non-null) phone number

## DFF

- Without integrity constraints in the database, maintaining data correctness requires:
  - Lots of users know what to do and do not make mistakes.
  - Dozens of programs correctly implement constraints in the code and stay up to date on changes.
- Implementing the constraints as part of the schema eliminates many issues.



# Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check (P)**, where P is a predicate



# Not Null Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**  
*name varchar(20) not null*  
*budget numeric(12,2) not null*



# Unique Constraints

- **unique** ( $A_1, A_2, \dots, A_m$ )
  - The unique specification states that the attributes  $A_1, A_2, \dots, A_m$  form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

# Simple Example

- Consider a simple example of an entity class *major*:
  - *major(id, name, track)*
  - “id” is a uniquely generated ID
  - “name” is the major name, e.g. “Computer Science,” “Economics,” ... ...
  - “track” is a sub-track/specialty within the major, e.g. “Applications,” “AI/ML,” ...
    - “track” is optional
    - The combination of *(name, track)* is *unique*.
- **Note:** In many DBMS, this automatically creates indices for keys/constraints.
- Switch to Notebook.

```
create table if not exists majors
(
 id int auto_increment
 primary key,
 major_name varchar(64) not null,
 major_track varchar(64) null,
 constraint table_name_pk
 unique (major_name, major_track)
);
```



# The check clause

- The **check** (P) clause specifies a predicate P that must be satisfied by every tuple in a relation.
- Example: ensure that semester is one of fall, winter, spring or summer

```
create table section
 (course_id varchar (8),
 sec_id varchar (8),
 semester varchar (6),
 year numeric (4,0),
 building varchar (15),
 room_number varchar (7),
 time_slot_id varchar (4),
 primary key (course_id, sec_id, semester, year),
 check (semester in ('Fall', 'Winter', 'Spring', 'Summer')))
```

DFF:

- We could handle the *semester check* with an *enum*.
- Switch to notebook for a slightly different example.



# Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If “Biology” is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for “Biology”.
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.



# Referential Integrity (Cont.)

- Foreign keys can be specified as part of the SQL **create table** statement  
**foreign key (*dept\_name*) references *department***
- By default, a foreign key references the primary-key attributes of the referenced table.
- SQL allows a list of attributes of the referenced relation to be specified explicitly.  
**foreign key (*dept\_name*) references *department* (*dept\_name*)**



# Cascading Actions in Referential Integrity

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- An alternative, in case of delete or update is to cascade

```
create table course (
 ...
 dept_name varchar(20),
 foreign key (dept_name) references department
 on delete cascade
 on update cascade,
 ...
)
```

- Instead of cascade we can use :
  - **set null**,
  - **set default**

## DFF:

- I do not like using *cascade*. I think making changes should be explicit.
- Other people disagree.
- You will get some simple practice on HW or exams.

# *Indexes*

## *Review, Concepts, Examples*



# 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.

# *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/... ...

## 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.

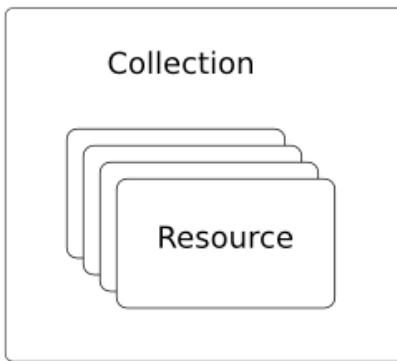
## Creating RESTful Webservice

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

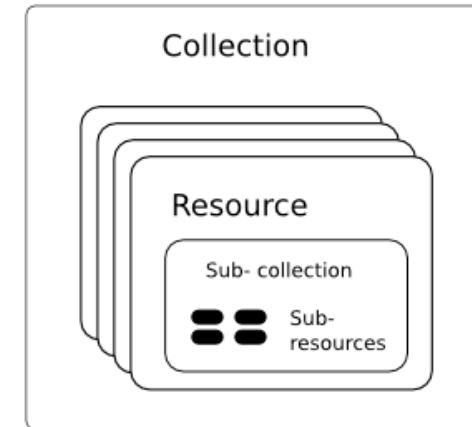
## Resource Model



A Collection with  
Resources

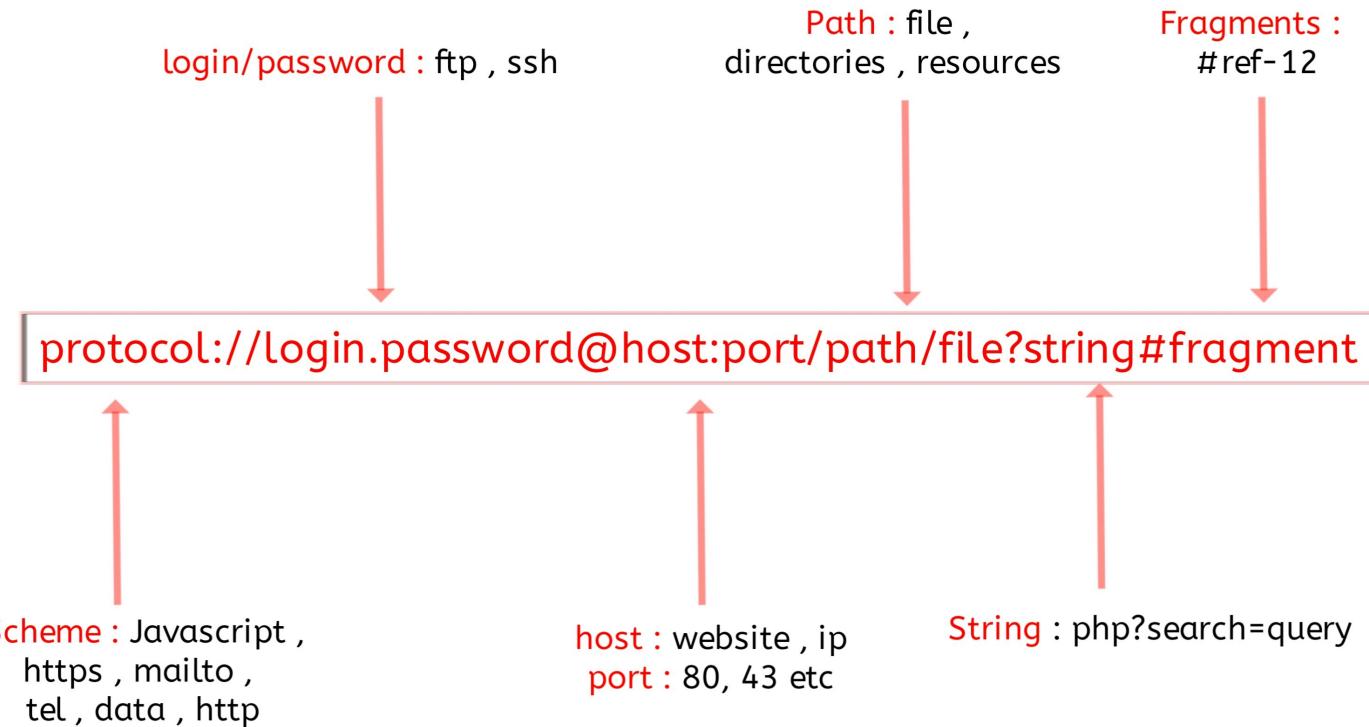


A Singleton  
Resource



Sub-collections and  
Sub-resources

# URLs



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

# 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<br>people row |
| GET         | /people/21                | SHOW KEYS FROM people ...;<br><br>SELECT * FROM people WHERE<br>playerID= 21                                     | people row                 |
| GET         | /people/21/batting        | SELECT batting.* FROM<br>people JOIN batting USING(playerID)<br>WHERE playerID=21                                |                            |
| GET         | /people/21/batting/2004_1 | SELECT batting.* FROM<br>people JOIN batting USING(playerID)<br>WHERE playerID=21<br>AND yearID=2004 AND stint=1 |                            |

# Application Architecture

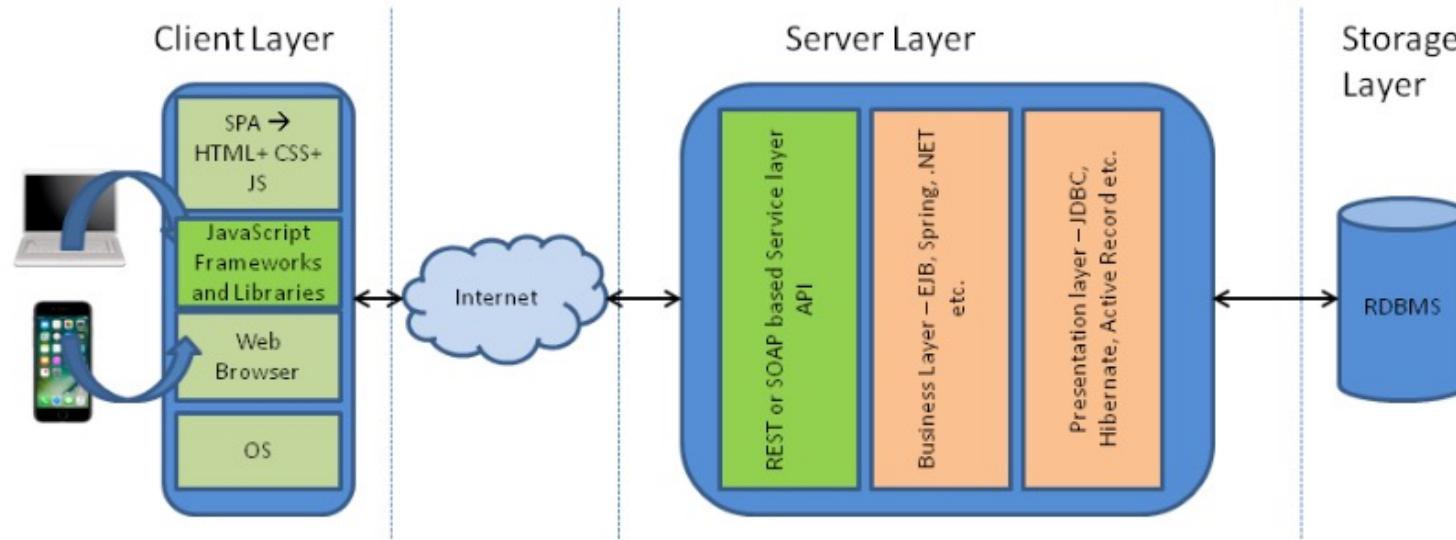


Diagram 2: The moving of the Web Layer from the Server to the Client

# Walkthrough

- Simple web application template.
- Calling some cloud APIs.

# *Sample Projects*

# Projects

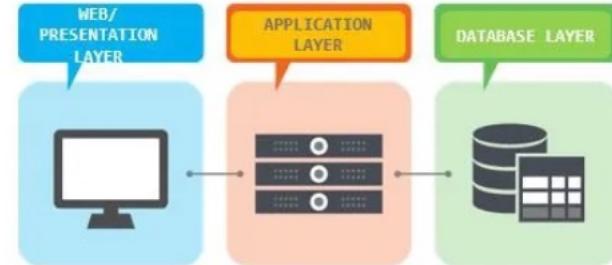
- The programming track will implement a simple, full stack web application.

## Full-stack Web Developer

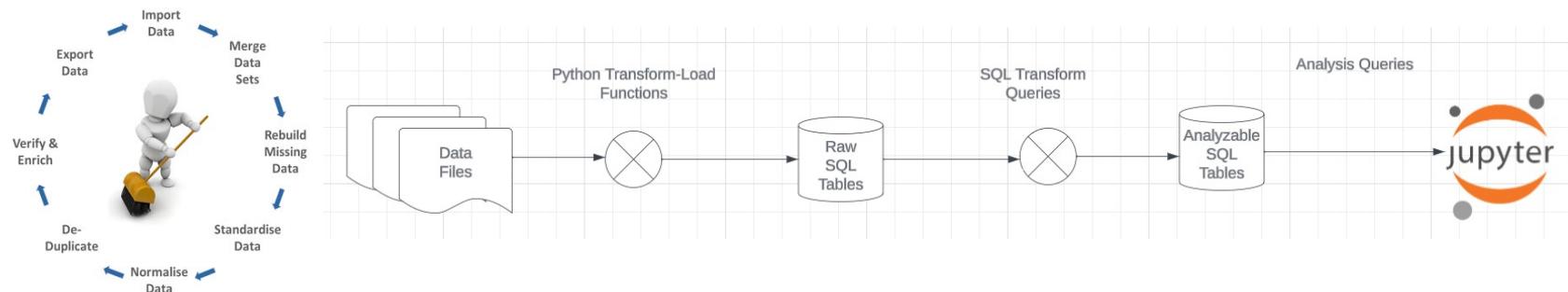
A full-stack web developer is a person who can develop both **client** and **server** software.

In addition to mastering HTML and CSS, he/she also knows how to:

- Program a **browser** (e.g. using JavaScript, jQuery, Angular, or Vue)
- Program a **server** (e.g. using PHP, ASP, Python, or Node)
- Program a **database** (e.g. using SQL, SQLite, or MongoDB)



- The non-programming track will implement a data engineering and visualization process in a Jupyter notebook.



# Game of Thrones

- Bottom-Up Data Mapping:
  - “Nouns” usually map to Entity/Entity Set.
  - Nouns inside other nouns often map to:
    - Attribute
    - Relationship
  - Verbs often map to relationships.
  - Adjectives usually map to properties.
- We will start with a subset of the information:
  - Game of Thrones:
    - Episodes
    - Characters
  - IMDB:
    - names\_basics
    - title\_basics
- Entities Sets
  - Character
  - Season
  - Episode
  - Scene
  - Location, Sublocation
  - ... ...
- Relationships
  - Character – Scene
  - Character – Character (e.g. KilledBy)
  - Season – IMDB Title
  - Character – IMDB Name
  - ... ...

# Game of Thrones

- Bottom-Up Data Mapping:

- IMDB: <https://developer.imdb.com/non-commercial-datasets/>

- Do not download.
    - Despite being “tiny” compared to the real world.
    - The datasets are too big for most laptops.

- We have info

- Game of Thrones: <https://github.com/jeffreylancaster/game-of-thrones>

- IMDB datasets
    - names\_basics
    - title\_basics

- Entities Sets

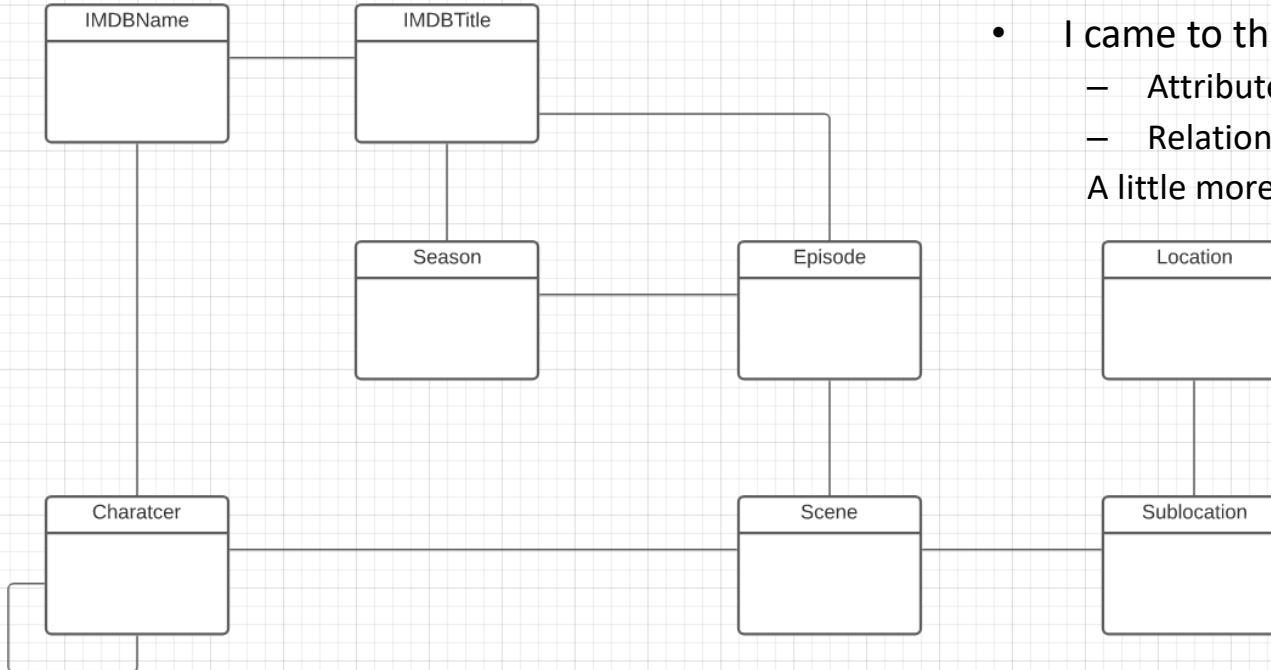
Character – IMDB Name

... ...

# Game of Thrones – Conceptual Model

Add shapes in Lucidchart ...

Add Entity Relationship Shapes



- With a little
  - Data exploration
  - Common sense
  - Judgment/experience
- I came to this conceptual model.
  - Attributes unspecified
  - Relationship required/cardinality unspecified.A little more exploration is needed.

# Sample Projects

- Walkthrough of Web Apps:
  - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/W4111-FastAPI-IMDB-Solution
  - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/current-dashboard
- Data Engineering:
  - /Users/donaldferguson/Dropbox/000/000-Columbia/2024-Spring/W4111-Intro-to-Databases-Spring-2024/Lectures/s-2024-Lecture-3/More-Data-Engineering.ipynb
  - And others.