# Types of Databases

## Relational Databases

A database where the data is stored in tables and different tables related to each other using primary and foreign keys.

* **Stored in Tables and rows**Relational databases like MySQL, PostgreSQL and SQLite3 represent and **store data in tables and rows**. They're based on a branch of algebraic set theory known as relational algebra.
* **Querying: SQL**Relational databases use Structured Querying Language (SQL), making them a good choice for applications that involve the management of several transactions. The structure of a relational database allows you to link information from different tables through the use of **foreign keys (or indexes)**, which are used to uniquely identify any atomic piece of data within that table. Other tables may refer to that foreign key, so as to create a link between their data pieces and the piece pointed to by the foreign key. This comes in handy for applications that are heavy into data analysis.
* If you want your application to handle a lot of complicated querying, database transactions and routine analysis of data, you’ll probably want to stick with a relational database. And if your application is going to focus on doing many database transactions, it’s important that those transactions are processed reliably. This is where ACID (the set of properties that guarantee database transactions are processed reliably) really matters, and where referential integrity comes into play.

### Advantages

* **Expressive query language & secondary Indexes.** Users should be able to access and manipulate their data in sophisticated ways to support both operational and analytical applications. Indexes play a critical role in providing efficient access to data, supported natively by the database rather than maintained in application code.
* **Strong consistency.** Applications should be able to immediately read what has been written to the database. It is much more complex to build applications around an eventually consistent model, imposing significant work on the developer, even for the most sophisticated engineering teams.
* **Enterprise Management and Integrations.** Databases are just one piece of application infrastructure, and need to fit seamlessly into the enterprise IT stack. Organizations need a database that can be secured, monitored, automated, and integrated with their existing technology infrastructure, processes, and staff, including operations teams, DBAs, and data analysts.

### Drawbacks

* DB **support only a few data types, doesn’t support inheritance** etc.
  + Object Relational Impedance Mismatch:
  + Data types mismatch
  + Inheritance
  + Associations: in OO language: pointer to data. In DB – foreign key to another table.
  + Data navigation: walking the object network (i.e. pointer) in OO language. In DB – need to use joins.
* **Scaling and distributing is limited**

## Non-Relational Databases

* **Stored in JSON objects:**Meanwhile, non-relational databases like MongoDB represent data in collections of JSON documents. The Mongo import utility can import JSON, CSV and TSV file formats. Mongo query targets of data are technically represented as BSON (binary JASON).

### Advantages

If your data model turns out to be very complex, or if you find yourself having to de-normalize your database schema, non-relational databases like Mongo may be the best way to go. Other reasons for choosing a non-relational database include:

* The need to store serialized arrays in JSON objects
* Storing records in the same collection that have different fields or attributes
* Finding yourself de-normalizing your database schema or coding around performance and horizontal scalability issues
* Problems easily pre-defining your schema because of the nature of your data model
* non-relational database is that your database is **not at risk for SQL injection attacks**, because non-relational databases don’t use SQL and are, for the most part, schema-less.
* Another major advantage, at least with Mongo, is that you can theoretically shard it forever (although that does bring up replication issues). Sharding distributes the data across partitions to overcome hardware limitations.
* **Flexible Data Model.** NoSQL databases emerged to address the requirements for the data we see dominating modern applications. Whether document, graph, key-value, or wide-column, all of them offer a flexible data model, making it easy to store and combine data of any structure and allow dynamic modification of the schema without downtime or performance impact.
* **Scalability and Performance.** NoSQL databases were all built with a focus on scalability, so they all include some form of sharding or partitioning. This allows the database to scale out on commodity hardware deployed on-premises or in the cloud, enabling almost unlimited growth with higher throughput and lower latency than relational databases.
* **Always-On Global Deployments**. NoSQL databases are designed for highly available systems that provide a consistent, high quality experience for users all over the world. They are designed to run across many nodes, including replication to automatically synchronize data across servers, racks, and data centers.

### Disadvantages

* **No joins** – you have to perform multiple queries in your code and join the data manually – can get ugly. Fast.
* **No automatic built-in transactions** - Since Mongo doesn’t automatically treat operations as transactions the way a relational database does, you must manually choose to create a transaction and then manually verify it, manually commit it or roll it back.

## Referential Integrity

Referential integrity is the concept in which multiple database tables share a relationship based on the data stored in the tables, and that relationship must remain consistent. This is usually enforced with cascading actions of adding, deleting and updating.

Keep in mind, there are three rules that referential integrity enforces:

1. We may not add a record to the Shelter Funding table unless the foreign key for that record points to an existing shelter in the Shelter table. You can think of this as a “No Unattended Child” rule or a “No Orphans” rule.
2. If a record in the shelter table is deleted, all corresponding records in the Shelter Funding table must also be deleted. The best way to handle this is by using **cascade delete**.
3. If the primary key for a record in the Shelter table changes, all corresponding records in the Shelter Funding (and other possible future tables with data relating to the Shelter table) must also be modified using something called a **cascade update**.

## Terms

**What is SQL?**

**Ans:** Structured Query language, SQL is an ANSI(American National Standard Institute) standard programming language which is designed specifically for storing and managing the data in the relational database management system (RDBMS) using all kinds of data operations.

Advantages of SQL:

* Simple SQL queries can be used to retrieve a large amount of data from the database very quickly and efficiently.
* SQL is easy to learn and almost every DBMS supports SQL.
* It is easier to manage the database using SQL as no large amount of coding is required.

**What is Database transaction?**

* **Ans:** Sequence of operation performed which changes the consistent state of the database to another is known as the database transaction. After the completion of the transaction, either the successful completion is reflected in the system or the transaction fails and no change is reflected.
* **ACID** ([***A****tomicity*](https://en.wikipedia.org/wiki/Atomicity_(database_systems))*,* [***C****onsistency*](https://en.wikipedia.org/wiki/Consistency_(database_systems))*,* [***I****solation*](https://en.wikipedia.org/wiki/Isolation_(database_systems))*,* [***D****urability*](https://en.wikipedia.org/wiki/Durability_(database_systems))) is a set of properties of [database transactions](https://en.wikipedia.org/wiki/Database_transaction) intended to guarantee validity even in the event of errors, power failures, etc. In the context of [databases](https://en.wikipedia.org/wiki/Database), a sequence of database operations that satisfies the ACID properties (and these can be perceived as a single logical operation on the data) is called a transaction

**What do you understand by Join?**

* **Ans:** Join is the process of explaining the relationship between different tables by combining columns from one or more table having common values in each. When a table joins with itself, it is known as Self Join

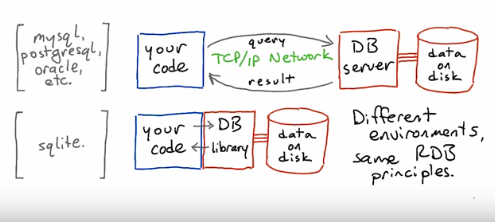
**Explain Primary Key and Composite Key.**

* **Ans: Primary key** is that column of the table whose every row data is uniquely identified. **Every row** in the table **must have a primary key** and **no two rows** can have **the same** primary key. Primary key value **can never be null nor can be modified or updated.**
* **Composite Key** is a form of the candidate key where a set of columns will uniquely identify every row in the table.

# Introduction

Data bases advantages over regular files:

* Persistent in disk (like files)
* **All data bases** support:
  + Allow reading and searching – much faster than files and much more space efficient.
  + To make it even faster, use indexing.
  + Allow multiple program and users to access and change the files at the same time without overwriting each other’s data.
* **Relational Databases** also support the following:
  + A very flexible query language for querying, summarizing and analysing the data
  + Allow defining and using constraints for protecting the consistency of the data.



# Terminology

**ORM** – Object-Relational Mapping – library that allows us to communicate with a data base using objects instead of direct SQL queries. This makes our code much more robust. Examples:

* + - **SQL Alchemy** – a python ORM for communicating with a database
    - **Entity** - .Net ORM framework
* Primary Key – A column that uniquely identifies the rows in a table. Sometimes there is a natural unique key (e.g. id number or country names).
* **Foreign Key** – a column(s) in one table that uniquely identifies rows in another table (like the serial id that is referred to in the other tables). Note: a foreign key doesn’t have to be unique in its own table (e.g. AnimalType in Food table)
* **Aggregation** – taking the data from multiple lines in the table and collating them into a single line.
* **Aggregation functions:** count (how many gorillas in the zoo), avg, max, min, sum

## Queries

* **create database** <database name> [options];
* **drop database** <database name> [options];
* **create table** <tablename> (  
   <column name> <column type> ,  
   <column name> <column type> ,….);
* **Select**: returns a table with the results:  
  **Select** <columns|\*(all) > **from** <table> **where** <conditions with and/or/not>  
  select name,birthdate from animals where species=”gorilla” and name=”Max”;  
  select <> from <> like
* **Limit** <count> [offset <skip>] – limit 10 offset 150 – return 10 rows starting with row 151
* **Order by** <columns> [Desc] – order by species, name – return ordered by species and then by names
* **Group by** <columns>,<**aggregate function**> **from** <table> where <conditions> **group by** <column> order by <columns>  
  select name, count(\*) as num from animals group by name;
* **Sub queries** – a sub-query that can be used instead of the table, the columns or the conditions:  
  select name, weight   
   from players   
   where weight < ( select avg(weight) as av from players );
* **View** – allows saving query: create view <viewname> as select…
* Update – change the contents of specific column(s) in specific row(s):  
  UPDATE <table> SET <column=value> WHERE <restriction>
* Insert – add a row with the specified values in the specified columns:  
  INSERT INTO <table> ( column1, column2, ... ) VALUES ( val1, val2, ... );
* DELETE – delete row(s) from a table:  
  DELETE FROM <table> WHERE <condition>
* Join – a regular join (inner) will only return rows that have values in both tables. Left-=join will also return left-only entries and right-join will also return right-only entries.
  + Full syntax:  
    SELECT animals.name , animals.species , diet.food  
    FROM animals JOIN diet ON animals.species = diet.species  
    WHERE food = ‘fish’
  + Short syntax:  
    SELECT animals.name , animals.species , diet.food  
    FROM animals, diet WHERE animals.species = diet.species  
    and food = ‘fish’

## Design

### Normalize Design

* Every row has the same number of columns
* Every column has only one value (e.g zoo diet table will have one raw for every food: Bear-fish, Bear-honey etc.)
* There is a unique key and everything in the row says something about this key. Note: the key can be one or more columns (e.g. in the zoo diet table – it is the whole row).
* Facts that don’t relate to the key belong in different table (e.g. items table with items and locations. The locations’ addresses belong in a different table).
* We can connect different tables with foreign keys.
* Tables shouldn’t imply relationships that don’t exist (e.g. person’t technology skills should be in a different table than their language skills).

### De-Normalizaing Tables

# AWS DynamoDB

## AWS

a secure cloud services platform, offering **compute power**, **database storage**, **content delivery** and other functionality to help businesses scale and grow.

# MongoDB

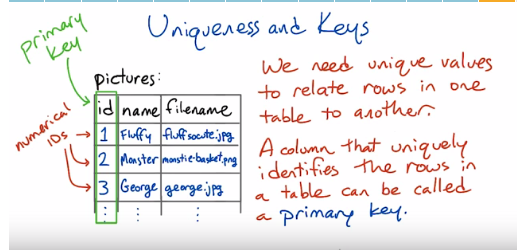
Document database (like DynamoDB)

<https://www.mongodb.com/try/download/community>

# Database Tables

* If we need to have multiple values in one of the column (brown bear eats fish and meat and plants), we’ll have as many lines we need so each of them will contain one value:  
  

## Table Keys



**Sometimes there’s a natural primary key** for example: country names. You have to make sure the natural key is really unique (example: there are many cities with the same name in the same states in the US).

## Operations on Databases

**Aggregation** – taking the data from multiple lines in the table and collating them into a single line.

**Aggregation functions:** count (how many gorillas in the zoo), avg, max, min, sum

# SQL

## Table Information

There is not standard for listing your tables and columns – every database does it in a different way:

* PostgreSQL: \dt and \d tablename
* MySQL: show tables and describe tablename
* SQLite: .tables and .schema tablename

## Data Types

Every database supports different types – see the manual for the specific database you use.

For PostgreSQL: <http://www.postgresql.org/docs/9.4/static/datatype.html>

Data types:

* text – a string of any length, like Python **str** or **unicode** types.
* integer – an integer value, like Python **int**.
* date – ‘<date>’ with single quotes around it. Otherwise, SQL will interpret this as int expression.  
  <data> - a calendar date; including year, month, and day.  
  Dates in our databases will always be in the international standard format, e.g. **'1999-12-31'**. Make sure to put single quotes around dates.
* **time** — a time of day.
* **timestamp** — a date and time together.
* **serial (in PostgreSQL)** – auto advancing serial number that starts from 1 by default.  
   if you want this value to be unique and primary key, you have to specify this   
   explicitly.

## Create and Drop

CREATE DATABASE <database name> [options];

DROP DATABASE <database name> [options];

CREATE TABLE <tablename> (  
 <column name> <column type> ,  
 <column name> <column type> ,….);

Examples:

CREATE TABLE distributors (

\*Unique\_name PRIMARY KEY

name varchar(40) DEFAULT 'Luso Films',

\*id integer serial,

\*modtime timestamp DEFAULT current\_timestamp

);

\* PRIMARY KEY ( column\_name [, ... ] ) – verifies unique and not null

\* id is serial - automatically generated value (by the DB).

\* modtime timestamp **DEFAULT** current\_timestamp – will automatically insert the current time to this box in the table.

CREATE TABLE films (

code char(5),

title varchar(40),

did integer **references** distributors (id), - this means that the did of any new value must be identical to an existing id in the distributors table. This helps us to maintain **referential integrity** of our data base – columns that are supposed to refer to each other are guaranteed to do so.

date\_prod date,

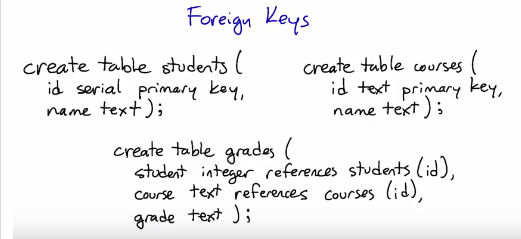
kind varchar(10),

len interval hour to minute,

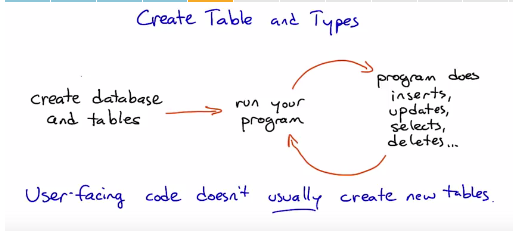
CONSTRAINT code\_title **PRIMARY KEY**(code,title) – multi-column primary key.

);

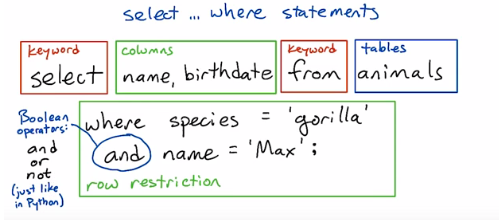
**Foreign Key** – a column(s) in one table that uniquely identifies rows in another table (like the serial id that is referred to in the other tables).

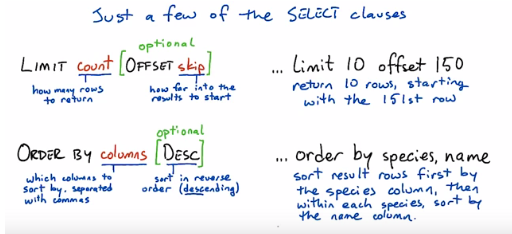
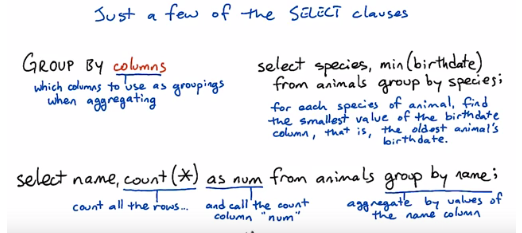


DROP TALBE <table name> [options];



## Select



* Database Query: **select** food **from** diet **where** species = ‘orangutan’  
  **will return a table with the results**.
* \* - selects all columns.
* Comparison operators: = , < , > , <= , >= , != , **like** ‘%<pattern>%’ (% is any string and <pattern> is what we’re actually looking for).
* The condition is a Boolean expression on column values. SQL supports the Boolean operations and, or, and not which work the same as in Python.
* Modifiers:
  + Limit, order:  
    
  + Group by (on aggregators only):
  + 

Examples:

* QUERY = "select max(name) from animals;"
* QUERY = "select \* from animals limit 10;"
* QUERY = "select \* from animals where species = 'orangutan' order by birthdate;"
* QUERY = "select name from animals where species = 'orangutan' order by birthdate desc;"
* QUERY = "select name, birthdate from animals order by name limit 10 offset 20;"
* QUERY = "select species, min(birthdate) from animals group by species;"
* QUERY = '''  
  select name, count(\*) as num   
  from animals group by name  
  **order by** num desc  
  limit 5;
* select species , count(\*) as num   
  from animals group by species   
  order by num desc limit 1;
* **select** animals.name, animals.species, diet.food, **count**(diet.food) **as** num  
  **from** animals, diet **where** animals.species = diet.species  
  **group by** diet.species  
  **having** num=1
* select diet.food  
  from animals, diet where animals.species = diet.species  
  group by diet.species  
  having count(diet.food)=1
* select ordernames.name , count(\*) as num  
  from animals , taxonomy , ordernames   
  where animals.species = taxonomy.name and taxonomy.t\_order = ordernames.t\_order  
  group by ordernames.name  
  order by num desc
* With explicit join style:  
  select ordernames.name, count(\*) as num  
  from (animals join taxonomy   
   on animals.species = taxonomy.name)  
   as ani\_tax  
   join ordernames  
   on ani\_tax.t\_order = ordernames.t\_order  
   group by ordernames.name  
   order by num desc
* Find pairs of roommates with no duplication from one table with columns: id-building-room.  
  select a.id, b.id, a.building, a.room  
   from residences as a, residences as b  
   where a.building = b.building  
   and a.room = b.room   
   and a.id != b.id   
   group by a.building , a.room - this will remove duplicate line (a.id,b.id and b.id,a.id with the same room)  
   order by a.building, a.room;
* select products.name, products.sku, count(sales.sku) as num  
   from products left join sales  
   on products.sku = sales.sku  
   group by products.sku;
* **Counting what’s not there** (with 0 entries in one of the tables):  
  select programs.name, count(bugs.filename) as num  
   from programs left join bugs  
   on programs.filename = bugs.filename  
   group by programs.name  
   order by num;
  + Something to watch out for: What do you put in the **count** aggregation? If you leave it as **count(\*)** or use a column from the **programs** table, your query will count entries that don't have bugs as well as ones that do.
  + In order to correctly report a zero for programs that don't have any entries in the **bugs** table, you have to use a column from the **bugs** table as the argument to **count**. For instance, **count(bugs.filename)** will work, and so will **count(bugs.description)**.

The most basic form of the **select** statement is to select a single scalar value:

**select** 2 + 2 **;**

More usefully, we can select one or more columns from a table.  With no restrictions, this will return all rows in the table:

**select** name, species **from** animals **;**

Columns are separated by commas; use **\*** to select all columns from the tables.

Quite often, we don't want all the data from a table.  We can restrict the rows using a variety of select clauses, listed below. There are also a wide variety of functions that can apply to columns; including aggregation functions that operate on values from several rows, such as **max** and **count**.

### where

The **where** clause expresses restrictions — filtering a table for rows that follow a particular rule. **where** supports equalities, inequalities, and boolean operators (among other things):

* **where species = 'gorilla'** — return only rows that have 'gorilla' as the value of the species column.
* **where name >= 'George'** — return only rows where the name column is alphabetically after 'George'.
* **where species != 'gorilla' and name != 'George'** — return only rows where species isn't 'gorilla' and name isn't 'George'.

### limit <to num of results>/ offset <skip over first number of results>

The **limit** clause sets a limit on how many rows to return in the result table. The optional **offset** clause says how far to skip ahead into the results. So **limit 10 offset 100** will return 10 results starting with the 101st.

### order by <columns> [desc]

The **order by** clause tells the database how to sort the results — usually according to one or more columns. So **order by species, name** says to sort results first by the species column, then by name within each species.

Ordering happens before limit/offset, so you can use them together to extract pages of alphabetized results. (Think of the pages of a dictionary.)

The optional **desc** modifier tells the database to order results in descending order — for instance from large numbers to small ones, or from Z to A.

### group by <columns>

The **group by** clause is only used with aggregations, such as **max** or **sum**. Without a **group by** clause, a select statement with an aggregation will aggregate over the whole selected table(s), returning only one row. With a **group by** clause, it will return one row for each distinct value of the column or expression in the **group by** clause.

SELECT column1, column2, <aggregate function on columns>

FROM table\_name

WHERE [ conditions ]

GROUP BY column1, column2

ORDER BY column1, column2

SELECT \* FROM `movies`

WHERE `category\_id` = 8

GROUP BY `category\_id`,`year\_released` ;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **movie\_id** | **title** | **director** | **year\_released** | **category\_id** |
| 9 | Honey mooners | John Schultz | 2005 | 8 |
| 5 | Daddy's Little Girls | NULL | 2007 | 8 |

### having

The **having** clause works like the **where** clause, but it applies after **group by** aggregations take place. The syntax is like this:

**select** columns **from** tables **group by** column **having** condition **;**

Usually, at least one of the columns will be an aggregate function such as **count**, **max**, or **sum** on one of the tables' columns. In order to apply **having** to an aggregated column, you'll want to give it a name using **as**. For instance, if you had a table of items sold in a store, and you wanted to find all the items that have sold more than five units, you could use:

**select name, count(\*) as num from sales having num > 5;**

You can have a **select** statement that uses only **where**, or only **group by**, or **group by** and **having**, or **where** and **group by**, or all three of them!

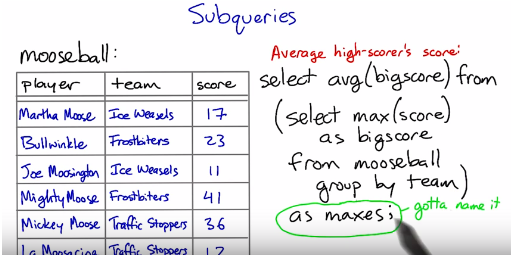
But it doesn't usually make sense to use **having** without **group by**.

If you use both **where** and **having**, the **where** condition will filter the rows that are going into the aggregation, and the **having** condition will filter the rows that come out of it.

You can read more about **having** here:

http://www.postgresql.org/docs/9.4/static/sql-select.html#SQL-HAVING

## Subselect – Subquery

  
<http://www.postgresql.org/docs/9.4/static/sql-expressions.html#SQL-SYNTAX-SCALAR-SUBQUERIES>  
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)

FROM states;

select name, weight from players where weight < (select avg(weight) as av from players);

<http://www.postgresql.org/docs/9.4/static/functions-subquery.html>

<http://www.postgresql.org/docs/9.4/static/sql-select.html#SQL-FROM>

## View

Allows us to save query

>> **create view** <viewname> **as** select ….

## Update

Changes the contents of a specific value in the table.

**UPDATE** <table> **SET** <column = value> **WHERE** <restrictions>

## Insert

**insert into** *table* **(** *column1, column2, ...* **) values (** *val1, val2, ...* **);**

If the values are in the same order as the table's columns (starting with the first column), you don't have to specify the columns in the **insert** statement:

**insert into** *table* **values (** *val1, val2, ...* **);**

For instance, if a table has three columns **(a, b, c)** and you want to insert into **a** and **b**, you can leave off the column names from the **insert** statement. But if you want to insert into **b** and **c**, or **a** and **c**, you have to specify the columns.

A single **insert** statement can only insert into a single table. (Contrast this with the **select** statement, which can pull data from several tables using a join.)

## Delete

**DELETE FROM** <table> **WHERE** <restrictions>

## Join

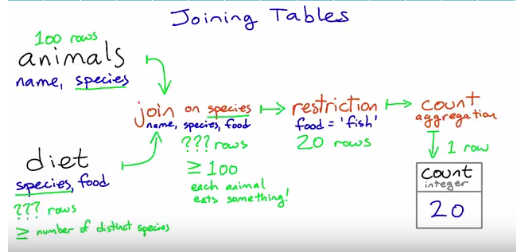
**Inner Join:** selects records that have matching values in both tables.

A regular (inner) join returns only those rows where the two tables have entries matching the join condition.   
A **left join** returns all those rows, plus the rows where the left table has an entry but the right table doesn’t.   
And a **right join** does the same but for the right table.

### Full Syntax:

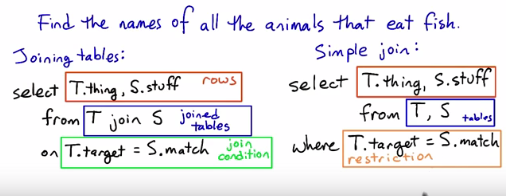
**select** animals.name , animals.species , diet.food  
**from** animals **join** diet **on** animals.species = diet.species  
**where** food = ‘fish’

\*\* The short style is more readable and easy.  
However, in some barebones databases (like SQLite) there can be a performance benefit to the explicit join style. But in PostgreSQL, the more server-oriented database system we'll be using next lesson, the query planner should optimize away any difference.



### Short Syntax:

**select** animals.name , animals.species , diet.food  
**from** animals , diet where animals.species = diet.species  
**and** food = ‘fish’



# SQLExpress and LocalDB

SQL Server Express is not recommended for use in production web applications. LocalDB in particular should not be used for production with a web application because it is not designed to work with IIS. However, a LocalDB database can be easily migrated to SQL Server or SQL Azure.

# PostgreSQL

## psql

Access the PostgreSQL (command line):

>> psql <database name>

The **psql** command-line tool is really powerful. There's a complete reference to it [in the PostgreSQL documentation](http://www.postgresql.org/docs/9.4/static/app-psql.html).

To connect **psql** to a database running on the same machine (such as your VM), all you need to give it is the database name. For instance, the command **psql forum** will connect to the **forum** database.

From within **psql**, you can run any SQL statement using the tables in the connected database. Make sure to end SQL statements with a semicolon, which is not always required from Python.

**\q** - exit psql.

**\c** <database> - connect to database.

You can also use a number of special **psql** commands to get information about the database and make configuration changes. The **\d posts** command shown in the video is one example — this displays the columns of the **posts** table.

Some other things you can do:

**\dt** — list all the tables in the database.

**\dt+** — list tables plus additional information (notably, how big each table is on disk).

**\H** — switch between printing tables in plain text vs. HTML.

Here's a fun one to run in **psql** while your forum web app is running:

**select \* from posts \watch**

(Note that **\watch** replaces the semicolon.) This will display the contents of the **posts** table and refresh it every two seconds, so you can see changes to the table as you use the app.

In order to do this, you'll need two terminal sessions into your VM — one running the forum app, and the other running **psql**. You can connect to the VM from any number of terminal windows at once — just open up another terminal, change to the **vagrant** directory, and type **vagrant ssh** again.

## psycopg2

psycopg2 is the python module that allow using PostgreSQL.

>>> import psycopg2

# Connect to an existing database

>>> conn = psycopg2.connect("dbname=test user=postgres")

# Open a cursor to perform database operations

>>> cur = conn.cursor()

Warning  
Never, **never**, **NEVER** use Python string concatenation (+) or string parameters interpolation (%) to pass variables to a SQL query string. Not even at gunpoint.

# Execute a command: this creates a new table

>>> cur.execute("CREATE TABLE test (id serial PRIMARY KEY, num integer, data varchar);")

# Pass data to fill a query placeholders and let Psycopg perform

# the correct conversion (no more SQL injections!)

>>> cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)",

... (100, "abc'def"))

# Query the database and obtain data as Python objects

>>> cur.execute("SELECT \* FROM test;")

>>> cur.fetchone()

# Make the changes to the database persistent

>>> conn.commit()

# Close communication with the database

>>> cur.close()

>>> conn.close()

**NOTES:**

* The correct way to pass variables in a SQL command is using the second argument of the [execute()](http://initd.org/psycopg/docs/cursor.html#cursor.execute) method:

>>> SQL = "INSERT INTO authors (name) VALUES (%s);" # Note: no quotes

>>> data = ("O'Reilly", )

>>> cur.execute(SQL**,** data) # Note: no % operator

* in order to verify that your code is protected against SQL injection attacks / bug – check that your database is safe from the following text (watch out – it will erase all the data from the database!):  
  '); delete from posts; --

# Design

### Normalize Design

A guide to Normalize Design: <http://www.bkent.net/Doc/simple5.htm>

In a normalized database, the relationships among the tables match the relationships that are really there among the data. Examples [here](https://www.udacity.com/course/viewer#%21/c-ud197/l-3490418600/m-3514018646) refer to tables in Lessons 2 and 4.

**1. Every row has the same number of columns.**   
In practice, the database system won't let us literally have different numbers of columns in different rows. But if we have columns that are sometimes empty (null) and sometimes not, or if we stuff multiple values into a single field, we're bending this rule.

The example to keep in mind here is the **diet** table from the zoo database. Instead of trying to stuff multiple foods for a species into a single row about that species, we separate them out. This makes it much easier to do aggregations and comparisons.

**2. There is a unique key and everything in a row says something about the key.**   
The key may be one column or more than one. It may even be the whole row, as in the **diet** table. But we don't have duplicate rows in a table.

More importantly, if we are storing non-unique facts — such as people's names — we distinguish them using a unique identifier such as a serial number. This makes sure that we don't combine two people's grades or parking tickets just because they have the same name.

**3. Facts that don't relate to the key belong in different tables.**   
The example here was the **items** table, which had items, their locations, and the location's street addresses in it. The address isn't a fact about the item; it's a fact about the location. Moving it to a separate table saves space and reduces ambiguity, and we can always reconstitute the original table using a **join**.

Also make sure you maintain **referential integrity** (when you define your database) of our data base – columns in different tables that are supposed to refer to each other are guaranteed to do so.

\*\* Identify for each table – which is/are its primary key(s), which are its foreign key(s). Should more than one column need to be the primary key (think duplication – **if we don’t allow multiple rows with the same values in these columns, they should be primary keys**).

**4. Tables shouldn't imply relationships that don't exist.**   
The example here was the **job\_skills** table, where a single row listed one of a person's technology skills (like 'Linux') and one of their language skills (like 'French'). This made it look like their Linux knowledge was specific to French, or vice versa ... when that isn't the case in the real world. Normalizing this involved splitting the tech skills and job skills into separate tables.

## De-Normalizing Tables

You will sometimes hear about de-normalization as an approach to making database queries faster by avoiding joins. This is an advanced topic beyond the scope of this course. But if you're interested in it, on modern database systems (such as PostgreSQL) it is often possible to meet the same goals using tools such as [indexes](http://www.postgresql.org/docs/9.4/static/sql-createindex.html) and [materialized views](http://www.postgresql.org/docs/9.4/static/sql-creatematerializedview.html).

# Programing

**ORM** – Object-Relational Mapping – allows us to communicate with a data base using objects instead of direct SQL queries. This makes our code much more robust.

## Python

### SQL Alchemy

**SQL Alchemy** – a python ORM for communicating with a database.

See examples under Sarit\Programming\Projects\WebDev\fullstack\_vm\vagrant\adoption\_center

Documentation:

<http://docs.sqlalchemy.org/en/latest/orm/tutorial.html>

<http://docs.sqlalchemy.org/en/rel_1_0/>

The SQLAlchemy Object Relational Mapper (**ORM**) presents a method of associating user-defined Python classes with database tables, and instances of those classes (objects) with rows in their corresponding tables. It includes a system that transparently synchronizes all changes in state between objects and their related rows, called a [unit of work](http://docs.sqlalchemy.org/en/latest/glossary.html#term-unit-of-work), as well as a system for expressing database queries in terms of the user defined classes and their defined relationships between each other.

**Query:**

veggie\_burgers = session.query(MenuItem).filter\_by(name = “Veggie Burger”)

for burger in veggie\_burgers:

print burger.price

**Update:**

* + - 1. Query:   
         burger\_to\_update = session.query(MenuItem).filter\_by(name = “Veggie Burger”).one()
      2. Update values (pure python):

burger\_to\_update.price += 10

* + - 1. Add the updated value back to the database (and commit):

session.add(burger\_to\_update)  
session.commit()

Delete:

* + - 1. Query:

burger\_to\_delete = session.query(MenuItem).filter\_by(name = “Veggie Burger”).one()

* + - 1. Delete (and commit):

session.delete(burger\_to\_delete)  
session.commit()

### relationship

# <http://docs.sqlalchemy.org/en/latest/orm/basic_relationships.html#relationship-patterns>

# SQL Cheat Sheet

|  |  |
| --- | --- |
| Command | Comments |
| create database <database name> [options];  drop database <database name> [options]; |  |
| create table <tablename> (  <column name> <column type> ,  <column name> <column type> ,….); |  |

# PSQL Cheat Sheet

|  |  |  |  |
| --- | --- | --- | --- |
| Command | Description | Usage | Action |
| psql | launches the psql command line interface | psql tournament | launches and connects to tournament database |
| Psql  --version | From the shell – get the version of the PostgreSQL |  |  |
| \c | connect | \c tournament | connects to the tournament database, drops connection to previous database |
| \i | import | \i tournament.sql | executes the sql commands within the sql file from psql |
| \? | help | \? | get help with psql commands |
| \q | quit | \q | quit the psql command line interface |
| \d | describe | \d matches | describes the table structure |
| \dt | list tables | \dt | list tables in current database |