DATA ANALYTICS REFERENCE DOCUMENT	
Document Title:	52553 - Applied Database
Document No.:	1548347488
Author(s):	Rita Raher, Gerhard van der Linde
Contributor(s):	

REVISION HISTORY

Revision	Details of Modification(s)	Reason for modification	Date	By
0	Draft release	52553 - Applied Database reference page	2019/01/24 16:31	Gerhard van der Linde

52553 - Applied Databases

Module Breakdown

- 40% continuous assessment
- 60% for a final project

I'll detail the breakdown of the 40% continuous assessment over the coming weeks in the module.

Week 1 - Introduction

What is data

- Datum
- Single piece of information fact or statistic.
- Data
    - A series of facts or statistics.
- Types of Data
    - Non digital information.
    - Digital Information
        - Active Digital Footprint
        - Passive Digital Footprint

Ever increasing data... per minute

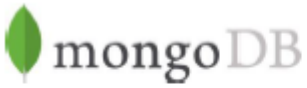
- 120+ new professionals join LinkedIn
- 456,000 tweets sent
- 3.6 million Google searches
- 4.1 million YouTube videos watched

- 18 million weather forecast requests received

Databases

- Relational Databases

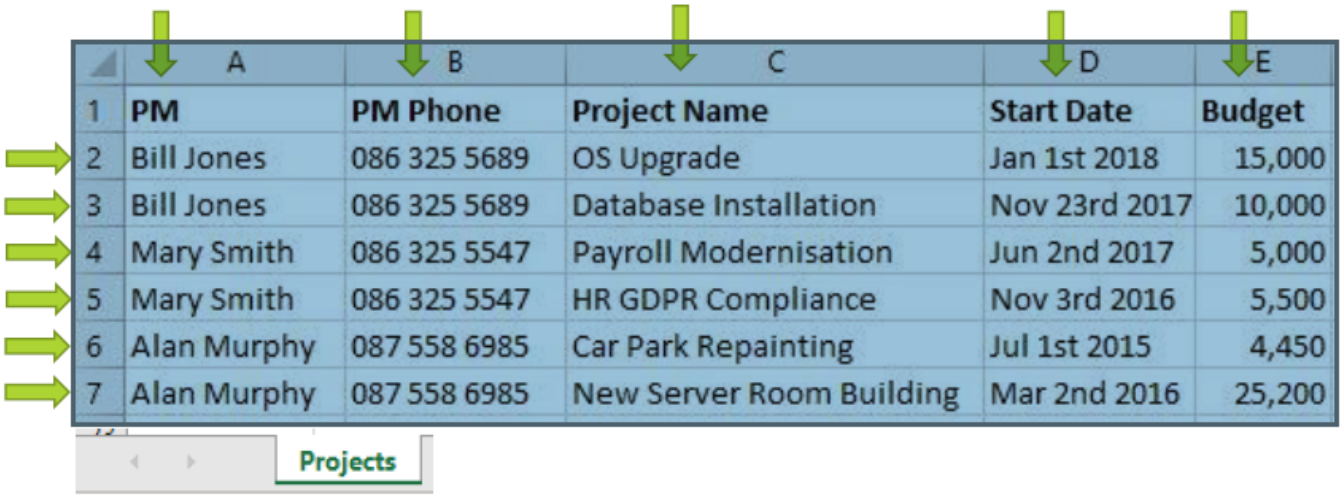
- Non-Relational (NoSQL) Databases



Relational Databases

- A relational database consists of a set of tables used for storing data.
- A table is collection of related data
- Each table has a unique name and may relate to one or more other tables in the database through common values.
- A table in a database is a collection of rows and columns. Tables are also known as entities or relations.
- A row contains data pertaining to a single item or record in a table. Rows are also known as records or tuples.
- A column contains data representing a specific characteristic of the records in the table. Columns are also known as fields or attributes.

Spreadsheets



	A	B	C	D	E
1	PM	PM Phone	Project Name	Start Date	Budget
2	Bill Jones	086 325 5689	OS Upgrade	Jan 1st 2018	15,000
3	Bill Jones	086 325 5689	Database Installation	Nov 23rd 2017	10,000
4	Mary Smith	086 325 5547	Payroll Modernisation	Jun 2nd 2017	5,000
5	Mary Smith	086 325 5547	HR GDPR Compliance	Nov 3rd 2016	5,500
6	Alan Murphy	087 558 6985	Car Park Repainting	Jul 1st 2015	4,450
7	Alan Murphy	087 558 6985	New Server Room Building	Mar 2nd 2016	25,200

Projects

Database Schema

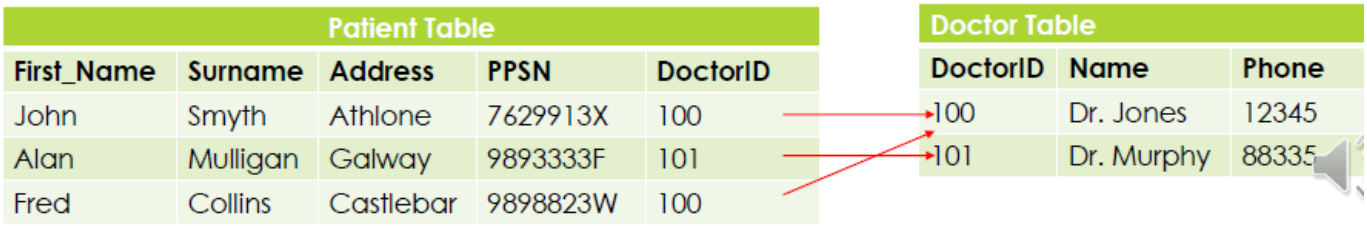
- A database consists of schemas, tables, views and other objects.
- A database schema represents the logical configuration of all or part of a database.
- It defines how the data, and relationships between the data, is stored
- Two types of Schema:
  - Physical Schema - Defines out how data is stored physically on a storage system in terms of files and indices.
  - Logical Schema - Defines the logical constraints that apply to the stored data, the tables in the database and the relationships between them.

Logical Schema

- The Logical Schema is designed before the database is created.
- No data is contained in the logical schema.

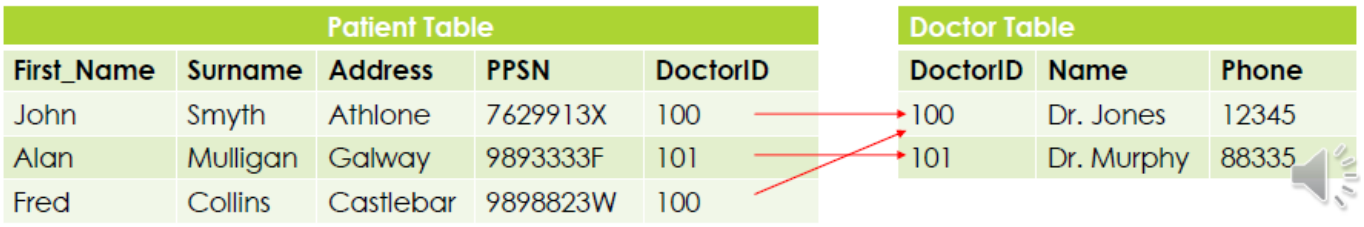
Patient Table
First_Name varchar(50)
Surname varchar(50)
Address varchar(200)
PPSN varchar(10)
Doctor varchar(50)
Doctor_Phone integer

Patient Table					
First_Name	Surname	Address	PPSN	Doctor	Doctor_Phone
John	Smyth	Athlone	7629913X	Dr. Jones	12345
Alan	Mulligan	Galway	9893333F	Dr. Murphy	88335
Fred	Collins	Castlebar	9898823W	Dr. Jones	12345



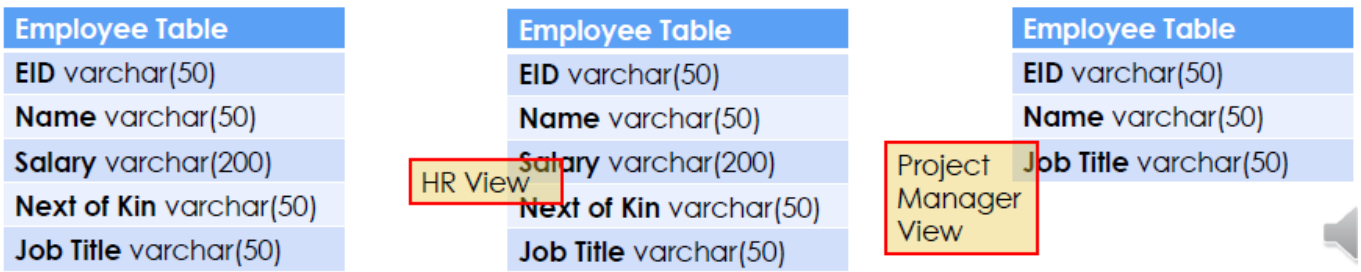
Spreadsheets vs Databases

	A	B	C	D	E	F
1	First Name	Surname	Address	PPSN	Doctor	Doctor Phone
2	John	Smyth	Athlone	7629913X	Dr. Jones	12345
3	Alan	Mulligan	Galway	9893333F	Dr. Murphy	88335
4	Fred	Collins	Castlebar	9898823W	Dr. Jones	12345



Database Management System (DBMS)

- A Database Management System (DBMS) is software for creating and managing databases.
- The DBMS interacts with the user, the database itself, and other systems in order to store, retrieve and process data.
- The DBMS provides a centralized view of data that can be accessed by multiple users, from multiple locations, in a controlled manner.
- The DBMS can limit what data the end user sees, as well as how that end user can view the data, providing many views of a single database schema.
- The DBMS provides data independence, freeing users (and application programs) from knowing where or how the data is stored. Any changes in how or where the data is stored is completely transparent due to the DBMS.
- CRUD (Create, Read, Update, Delete) functions



DBMS Functions

- Data Storage Management
- Security
- Backup and Recovery
- Transaction Management
- Debit Customer a/c
- Update Shipping Table
- Update Products Table
- Credit Store a/c
- Data integrity
- Concurrency

▶ Data integrity

Patient Table
First_Name varchar(50)
Surname varchar(50)
Address varchar(200)
PPSN varchar(10)
DoctorID integer

Doctor Table
DoctorID integer
Name varchar(50)
Phone integer

Patient Table				
First_Name	**Surname**	**Address**	**PPSN**	**DoctorID**
John	Smyth	Athlone	7629913X	100
Alan	Mulligan	Galway	9893333F	101
Fred	Collins	Castlebar	9898823W	100
Mary	Connolly	Tuam	6789932A	200

Doctor Table		
DoctorID	**Name**	**Phone**
100	Dr. Jones	12345
101	Dr. Murphy	88335
Dr. Kane	Dr. Kane	2314

Advantages of DBMSs

- Controlling Redundancy
- Data Integrity
- Enforcement of Standards
- Backup and Recovery
- Security

▶ Controlling Redundancy

Instead of each application having its own files with data stored multiple times, a centralised DBMS can store it once and allow many users to access it eliminating duplication.

	A	B	C	D	E	F
1	**First Name**	**Surname**	**Address**	**PPSN**	**Doctor**	**Doctor Phone**
2	John	Smyth	Athlone	7629913X	Dr. Jones	12345
3	Alan	Mulligan	Galway	9893333F	Dr. Murphy	88335
4	Fred	Collins	Castlebar	9898823W	Dr. Jones	12345

Patient Table				
First_Name	**Surname**	**Address**	**PPSN**	**DoctorID**
John	Smyth	Athlone	7629913X	100
Alan	Mulligan	Galway	9893333F	101
Fred	Collins	Castlebar	9898823W	100

Doctor Table		
DoctorID	**Name**	**Phone**
100	Dr. Jones	12345
101	Dr. Murphy	88335

Disadvantages of DBMSs

- Complexity
- Size
- Performance
- Higher impact of failure

Week 2 - Getting Info from Databases

topic_2_-_lecture.pdf

SQL

- Structured Query Language
- Standard Relational Database Language
- SQL is an ANSI/ISO standard, but different databases e.g. MySQL, SQL Server, Oracle may use their own proprietary extensions on top of the standard SQL.

What can SQL do?

CRUD

- Create a new database
- Create tables in a database
- Insert data into a database
- Read data from a database
- Update data in a database
- Delete data from a database
- Manage transactions
- Manage concurrency
- Backup and recovery
- Manage users

SQL vs MySQL

- SQL is a language.
- MySQL is a database management system.

Creating a database

```
mysql> create database myFirstDatabase;
Query OK, 1 row affected (0.01 sec)
```

```
mysql> CREATE dataBase MYFirstDATABASE;
Query OK, 1 row affected (0.01 sec)
```

```
mysql> create
```

```
    -> database
    -> MyFirstDatabase
    -> ;
Query OK, 1 row affected (0.01 sec)
```

Using a Database

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| myfirstdatabase    |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)
```

```
mysql> use myfirstdatabase;
Database changed
```

Creating Tables

MySQL Data Types

Car Attributes

- Make - Varchar(20)
- Model - Varchar(20)
- Registration - Varchar(20)
- Colour - Varchar(20)
- Mileage - Integer
- Engine Size - float(2,1)
- Cylinders
- Crankshaft

```
mysql> create table car (
    -> make VARCHAR(20),
    -> model VARCHAR(20),
    -> registration VARCHAR(15),
    -> colour VARCHAR(10),
    -> mileage INTEGER,
    -> enginSize FLOAT(2,1));
Query OK, 0 rows affected (0.09 sec)
```

Describing Tables

```
mysql> describe car;
+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
| make         | varchar(20) | YES  |     | NULL    |       |
| model        | varchar(20) | YES  |     | NULL    |       |
```

```
| registration | varchar(15) | YES  |     | NULL    |       |
| colour       | varchar(10) | YES  |     | NULL    |       |
| mileage      | int(11)     | YES  |     | NULL    |       |
| enginSize    | float(2,1)  | YES  |     | NULL    |       |
+--------------+-------------+------+-----+---------+-------+
6 rows in set (0.02 sec)
```

Primary key

Primary key - unique values. 1 primary key pair table add personID - int auto_increment flag to increase

Using reg as primary key

```
reg VARCHAR(15)
PRIMARY KEY(reg)
```

SELECT

```
SELECT name FROM person;

SELECT * FROM person;
```

WHERE

```
SELECT name FROM person where NOT isStudent

SELECT name
 FROM person
WHERE isStudent
AND sex = “M”;
```

where >=, <=, Between

```
SELECT personID, NAME, AGE
FROM person
WHERE age >= 20
AND age <= 39;

SELECT personID, NAME, AGE
FROM person
WHERE age between 20 and 39;
```

Like

```
SELECT personID, NAME, AGE
FROM person
WHERE name LIKE “%a%”;

SELECT personID, NAME, AGE
FROM person
WHERE name LIKE “_a%”;
```

IN

```
SELECT personID, NAME, AGE
FROM person
WHERE  age = 12
OR age = 13
OR age = 14
OR age = 15;
```

AND OR

```
SELECT personID, NAME, AGE
FROM person
WHERE  age  IN
 (12, 13, 14, 15);

SELECT NAME, AGE
FROM person
WHERE  sex =“M”
AND name LIKE “S%”
OR name LIKE “A%"
```

Limit

```
SELECT NAME, AGE
FROM person
WHERE  sex =“M”
AND name LIKE “S%”
OR name LIKE “A%"
LIMIT 1;


SELECT NAME, AGE
FROM person
WHERE  sex =“M”
AND name LIKE “S%”
OR name LIKE “A%"
LIMIT 0,3;
```

DISTINCT

```
SELECT DISTINCT(name)
```

order by name DESC

YEAR() DAY()

```
SELECT name, age, MONTHNAME(dob)
FROM person
WHERE day(DOB) between 1 and 11
AND name NOT like “A%"
order by name DESC
```

Import a database

```
use <database>
show tables;

drop database <name>;

describe <database>;
select * from cars;
```

Week3 - Applied Databases MySQL Functions and Procedures

Functions

- Mysql can do more than store and retrieve data
- It can also manipulate the data before storing or retrieving it, via functions.
- A function is a piece of code that performs some operation and returns a result.
- Some functions accept parameters, others do not

Built-in functions

- String functions https://dev.mysql.com/doc/refman/8.0/en/string-functions.html
- Numeric functions https://dev.mysql.com/doc/refman/8.0/en/numeric-functions.html
- Date and time functions https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html
- Aggregate functions https://dev.mysql.com/doc/refman/8.0/en/group-by-functions-and-modifiers.html
- MySql Information Functions https://dev.mysql.com/doc/refman/8.0/en/information-functions.html
- MySQL Control Flow Functions https://dev.mysql.com/doc/refman/8.0/en/control-flow-functions.html

String Functions

- **Upper()**
  - Returns an uppercase version of a string
- **STRCMP()**: Compares two strings and returns:
  - 0 if string 1 - string 2
  - -1 if string 1 < string 2
  - 1 if string 1 > string 2
- **ASCII()**
  - Returns the ASCII value of the first character in a string
- **REPLACE(string, from_string, to_string)**
  - Replaces all occurrences of s substring within a string, with a new substring
    - string - The original string
    - from_string - The substring to be replaced
    - to_string - The new replacement string

```
REPLACE(name, “Ms”, “MRS”)
```

- **SUBSTR(string, start, length)**
  - extract a substring from a string
    - string - the start to extract from

    - start - The start position within the string
    - length - The number of character to be extracted

```
SUBSTR(name, 1, 3)**
```

- **SQRT(Number)**
    - Returns the square root of a number
- **Round(number, decimals)**
    - Rounds a number to a specified number of decimal places

```
ROUND(enginesize)
```

- **DATEDIFF(date1, date2)**
    - returns the number of days between the 2 dates
- **DATE_FORMAT(date, format)**
    - formats a date

```
DATE_FORMAT(dob, "%d-%m-%y")
```

Aggregate Functions

An aggregate function performs a calculation on a set of values and returns a single value.

- **AVG()**
- **MIN()**
- **MAX()**
- **SUM()**
- **COUNT()**

Group By The groupie statement is often used with aggregate functions(COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns

```
SELECT level, AVG(experience)
FROM teacher GROUP BY level;

SELECT ROUND(AVG(mileage)) as "KMs"
FROM car;

SELECT model,  ROUND(AVG(mileage)) as "KMs"
FROM car
GROUP BY model;


SELECT model,  ROUND(AVG(mileage)) as "KMs"
FROM car
WHERE mileage > 60000
GROUP BY model;
```

HAVING

- The HAVING clause is often used with the GROUP BY clause to filter groups based on a specified condition.
- if the GROUP BY clause is omitted, the having clause behaves like the WHERE clause.
- The HAVING clause applies a filter condition to each group of rows
- The WHERE clause applies the filter condition to each initial row.

```
SELECT model,  ROUND(AVG(mileage)) as "KMs"
FROM car
WHERE mileage > 60000
GROUP BY model
```

```
HAVING KMs > 250000;
```

Information Functions

DATABASE()
USER()

```
select DATABASE();
select USER();
```

Control Flow Functions

- IF(condition, value_if_true, value_if_false)
  - condition - value to test
  - value_if_true - Value to return if condition is True
  - value_if_false - Value to return if condition is False

```
SELECT IF(150>200, “yes”, “no”)”T/F”;
```

```
SELECT *,IF(experience >= 20 AND experience <= 45, “Y”, “”) as “Payrise Due”
FROM teacher
```

```
CASE WHEN condition 1 THEN result 1
    WHEN condition 2 THEN result 1
    WHEN condition n THEN result n
    ELSE result
END
```

```
SELECT name, dob,
CASE
    WHEN month(dob) in (2,3,4) THEN “Spring”
    WHEN month(dob) in (5,6,7) THEN “Summer”
    WHEN month(dob) in (8,9,10) THEN “Autumn”
    WHEN month(dob) in (11,12,1) THEN “Winter”
END as Season
from person;
```

```
SELECT name, dob,
CASE
    WHEN month(dob) in (2,3,4) THEN “Spring”
    WHEN month(dob) in (5,6,7) THEN “Summer”
    ELSE “"
END as Season
from person;
```

Stored Routines

A stored routine is user-written code that extends the functionality Mysql.

Uses

- When multiple client apps are written in different languages or work on different platforms, but need to perform the same database operations.
- To ensure security. Applications cannot directly access tables only stored routines.

Advantages

- Speed
  - Performance of applications accessing the database is increased.
  - This is because stored procedures are complied and stored in the database
- Traffic
  - Instead of sending multiple lengthly SQL statements, the application has to send only the name and parameters of the stored routine.

Disadvantages

- Complexity
- Not designed for complex business logic
- Difficult to debug. Only a few database management systems allow to debug stored procedure. MySQL is not one of them.
- Performance. A DBMS is not well-designed for logical operations.

MySQL Stored Functions

- A stored fountain is a special kind stored routine that returns a single value.
- Stored functions are used to encapsulate common formulas or business logic rules that are reusable among Sql statement or stored routine.
- A functions take 0 or more input parameters and return a single value

https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html

```
CREATE FUNCTION add2Nums(num1 integer, num2 integer)
RETURNS integer
DETERMINISTIC
BEGINS
    RETURN num1 + num2;
END

SELECT add2Nums(3, 10);
```

```
CREATE FUNCTION discount(age INT(11))
RETURNS VARCHAR(3)
DETERMINISTIC
BEGIN
    IF age < 16 THEN
        RETURN “0%”;
    ELSEIF age < 26 THEN
        RETURN “10%”;
    ELSEIF age < 40 THEN
        RETURN “20%”;
    ELSEIF age < 60 THEN
        RETURN “30%”;
    ELSE
        RETURN “40%”;
    END IF;
END
```

```
SELECT name, age, discount(age) “Discount”
FROM person
```

Stored Procedures

FUNCTIONS	PROCEDURES
Return a single value	Retun 0 or more values
Only select	select, insert, update, delete

FUNCTIONS	PROCEDURES
Cant use stored procedures	can use stored functions
Does not support transactions	Support transactions

https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html

```
Create procedure
CREATE PROCEDURE make_mileage(mk VARCHAR(20), ml INT(11))
DETERMINISTIC
BEGIN
    SELECT * FROM CAR
    WHERE make LIKE mk
    AND mileage < ml
    ORDER BY mileage;
END
```

Call Procedure

```
call make_mileage(“Toyota”, 200000)
call make_mileage(“Ford”, 5000)
```

MySQL Routine Management

Finding Functions and Procedures

```
SELECT name, type from MYSQL.PROCE limit 3;
```

What’s in a function Procedure

```
SHOW CREATE FUNCTION add2nums;
```

Drop a Function or Procedure

```
DROP FUNCTION add2nums
```

Week 3 Excercises

Week 4 - Normalisation

topic_4_-_normalisation.pdf

Normalization is the process of organizing the columns (attributes) and tables (relations) of a relational database to minimize data redundancy.

show create table

```
show create table manufacturer;
+--------------+-------------------------------------+
| Table        | Create Table                        |
```

```
+--------------+-------------------------------------+
| manufacturer | CREATE TABLE `manufacturer` (
  `manu_code` varchar(3) NOT NULL,
  `manu_name` varchar(200) NOT NULL,
  `manu_details` varchar(400) DEFAULT NULL,
  PRIMARY KEY (`manu_code`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+--------------+-------------------------------------+
1 row in set (0.00 sec)
```

Get info from multiple tables

- INNER JOIN
- LEFT JOIN

INNER JOIN	LEFT JOIN
Return rows from two tables only when the JOIN condition is met.	Retrun rows from two tables when the JOIN condition is met.
If JOIN condition is **not** met, nothing is returned from either table.	If JOIN condition is **not** met, rows from the first (LEFT) table are returned and NULL is returned instead of rows from the second table.

Functions needed for exercises

```
select substring()
```

```
mysql> SELECT SUBSTRING('Quadratically',5);
    -> 'ratically'
mysql> SELECT SUBSTRING('foobarbar' FROM 4);
    -> 'barbar'
mysql> SELECT SUBSTRING('Quadratically',5,6);
    -> 'ratica'
mysql> SELECT SUBSTRING('Sakila', -3);
    -> 'ila'
mysql> SELECT SUBSTRING('Sakila', -5, 3);
    -> 'aki'
mysql> SELECT SUBSTRING('Sakila' FROM -4 FOR 2);
    -> 'ki'
```

topic_4_exercises.pdf

Topic 5 - Insert Update Delete

topic_5_-_insert_update_delete.pdf

Insert

- CRUD (Create/Insert, Read, Update Delete)
  https://dev.mysql.com/doc/refman/8.0/en/insert.html
- INSERT INTO <table> VALUES (value1, value2, valueN);
- INSERT INTO <table> (column1, column2, column)
  VALUES (value1, value2, valueN);

```
INSERT INTO person VALUES(1, "John", 23, "M", 1);
```

```
mysql> SELECT * FROM person;
+----------+------+------+------+-----------+
| personID | name | age  | sex  | isStudent |
+----------+------+------+------+-----------+
|        1 | John |   23 | M    |         1 |
+----------+------+------+------+-----------+
1 row in set (0.00 sec)
```

```
INSERT INTO person (age, sex, isStudent) VALUES (20, "F", 0);
```

https://dev.mysql.com/doc/refman/8.0/en/working-with-null.html

Update

https://dev.mysql.com/doc/refman/8.0/en/update.html

- UPDATE <table> SET column1 = value1, columnN, valueN;
- UPDATE <table> SET column1 = value1, columnN, valueN
  WHERE condition;

```
UPDATE Person
  SET age = 23
  WHERE personID = 3;

UPDATE person
  set name = CONCAT(IF(sex="M", "MR.", "Ms.",name));
```

Delete

https://dev.mysql.com/doc/refman/8.0/en/delete.html * DELETE FROM <table>; * DELETE FROM <table>
WHERE condition;

```
DELETE from PERSON
  WHERE personID = 6;

DELETE FROM person
  WHERE sex = "M"
  AND isStudent
```

```
  AND age > 20;
```

Foreign Keys

Foreign keys can be used to define table behavior when data is deleted. The default behavior for MySQL is ON DELETE RESTRICT, even if not specified.

- ON DELETE RESTRICT
- ON DELETE CASCADE
- ON DELET SET NULL

MySQL Command	MySQL Behaviour
RESTRICT	Prevents entried from being deleted where a foreign key exists
CASCADE	Proceed to delete entries from the table worked on as well as deleting the referenced entries
SET NULL	Delete the entries after setting the references to NULL

READ using SubQueries

```
SELECT emp_no, first_name, last_name
  FROM employees
  WHERE emp_no IN(
    SELECT emp_no
    FROM salaries
    WHERE salary = (
      SELECT MAX(salary)
      FROM salaries
    )
  );
```

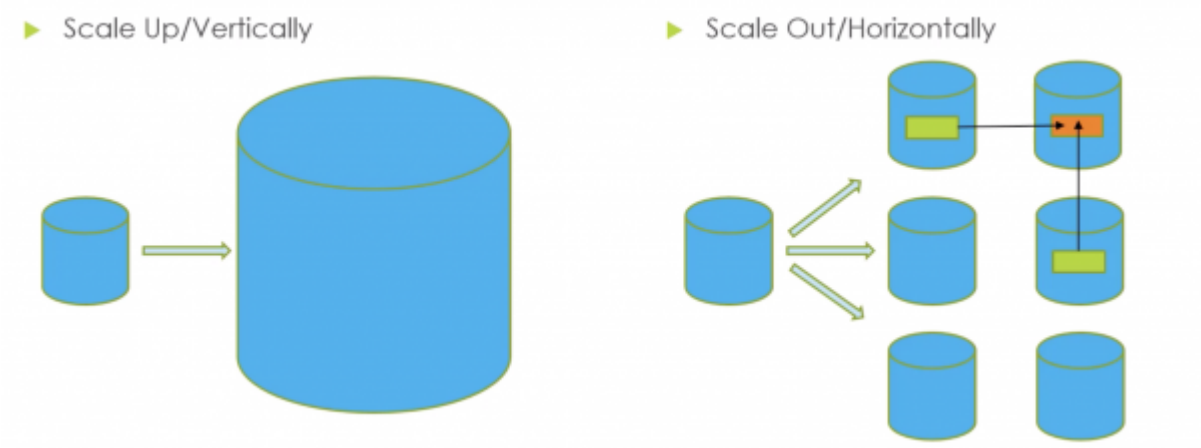
Topic 6 - MongoDB I

Why NoSQL Databases?

Scalability

Scale Up/Vertically: means moving the database to a bigger server.

Scale Out/Horizontally



Unstructured Data

```
CustomerID INTEGER
Name VARCHAR(20)
Phone VARCHAR(20)
Address VARCHAR(50)
Email VARCHAR(50)
Twitter VARCHAR(50)
```

CustomerID*	Name	Phone	Address	Email	Twitter
100	John	086 3304896	Tuam, Co. Galway	John@gmail.com	@John123
101	Alan	NULL	Athenry, Co. Galway	NULL	NULL
102	Mary	091 5688874	Galway, Co. Galway	Mary@yahoo.com	NULL
103	Tom	090 6458959	Athlone, Co. Westmeath	NULL	NULL
104	Alice	094 1245763	Castlebar, Co. Mayo	NULL	@AliceC1965

Add on new features later on like email and then twitter etc...

MongoDB

- Document Database
- Schemaless
- Horizontal Scalability Through sharding[1)]
- Duplication of data

JSON

- JSON - JavaScript Object Notation
- Lightweight data-interchange format
- Machine/Human readable
- Language independent
- JSON Structure
    - Name/Value pair
    - Ordered Lists

JSON Datatypes

Number

```
{
  "id" : 1
}
```

```
{
  "id" : 3.14
}
```

Note that there is no distinction between integer and floating point numbers.

String

```
{
  "id" : 1,
  "fname" : "John"
}
```

Boolean

```
{
  "reg" : "09-G-13"
  "hybrid" : false
}
```

Array

```
{
  "student" : "G00257854"
```

```
  "subjects" : ["Databases", "Java", "Mobile Apps"]
}
```

Object Document

```
{
  "student" : "G00257854"
  "address" : {
    "street" : "Castle Street"
    "town" : "Athenry"
    "county" : "Galway"
  }
}
```

JSON USES


```
{
  "type": "FeatureCollection",
  "totalFeatures": 28,
  "features": [
    {
      "type": "Feature",
      "id": "MINES_SiteLocation.fid-48245d43_1693a57810c_6682",
      "geometry": {
        "type": "MultiPoint",
        "coordinates": [
          [
            -6.78797543,
            54.15895923
          ]
        ]
      },
      "geometry_name": "Shape",
      "properties": {
        "Name": "Monaghan Pb",
        "Code": "MON",
        "SiteLocation": "Monaghan",
        "X_Centroid": 279389,
        "Y_Centroid": 323403,
        "CommodityProduced": "Pb(Zn-Ba-Ag),Sb",
        "URL": "No report available",
        "URLtext": "Link to More Information",
        "Description": "The Monaghan lead mines are made up of thi
      }
    },
    {
      "type": "Feature",
      "id": "MINES_SiteLocation.fid-48245d43_1693a57810c_6683",
      "geometry": {
```


https://data.gov.ie/dataset/mines-site-district/resource/8920e026-a3e7-4987-a9fe-

```
{
    status: "ok",
    totalResults: 10,
    articles: [
        {
            source: {
                id: "national-geographic",
                name: "National Geographic"
            },
            author: "Sarah Gibbens, Laura Parker",
            title: "Creatures in the deepest trenches of the sea are ea
            description: "In six of the ocean's deepest crevasses, scie
            shrimp-like creatures chomping on tiny bits of plastic.",
            url: https://www.nationalgeographic.com/environment/2019/02
            mariana-trench-eat-plastic.html,
            urlToImage:
            https://www.nationalgeographic.com/content/dam/environment/
            ,
            publishedAt: "2019-03-01T09:37:54.1275978Z",
            content: null
        },
        {
            source: {
                id: "national-geographic",
                name: "National Geographic"
            },
            author: "National Geographic Staff",
            title: "See the top 10 pictures entered in our Instagram co
            description: "Instagram users submitted more than 94,000 ph
```


https://newsapi.org/s/national-geographic-api

MongoDB, JSON and BSON

- JSON object = MongoDB document
- Internally, MongoDB represents JSON documents in binary-encoded format called BSON (Binary JavaScript Object Notation)
- BSON extends JSOM model to provide additional data types as well as indexes.

MongoDB Structures

Document  - slide 12....

A document is record in a MongoDB collection and the basic unit of data in MongoDB. Documents are analogous to JSON objects or records in an RDBMS.

```
{
    "_id"    : ObjectId("5919fecf0822ef8ecec132f8"),
    "name"   : "John",
    "house"  : 31,
    "street" : "Main St.",
    "town"   : "Athenry"
}
```

Collection

- A grouping of MongoDB documents.
- Collections are analogous to RDBMS tables.
- A collection exists within a single database.
- Collections do not enforce a schema. Documents within a collection can have different fields.
- Typically, all documents in a collection have a similar or related purpose.

```
{
    "_id"    : ObjectId("5919fecf0822ef8ecec132f8"),
    "name"   : "John",
    "house"  : 31,
    "street" : "Main St.",
    "town"   : "Athenry",
    "county" : "Galway"
}

{
    "_id"      : ObjectId("591a000f0822ef8ecec132f9"),
    "name"     : "Alan",
    "townland" : "Litirmor",
    "town"     : "Ballymurphy",
    "county"   : "Cork"
}
```

Database A number of databases can be run on a single MongoDB server.

MongoDB Commands

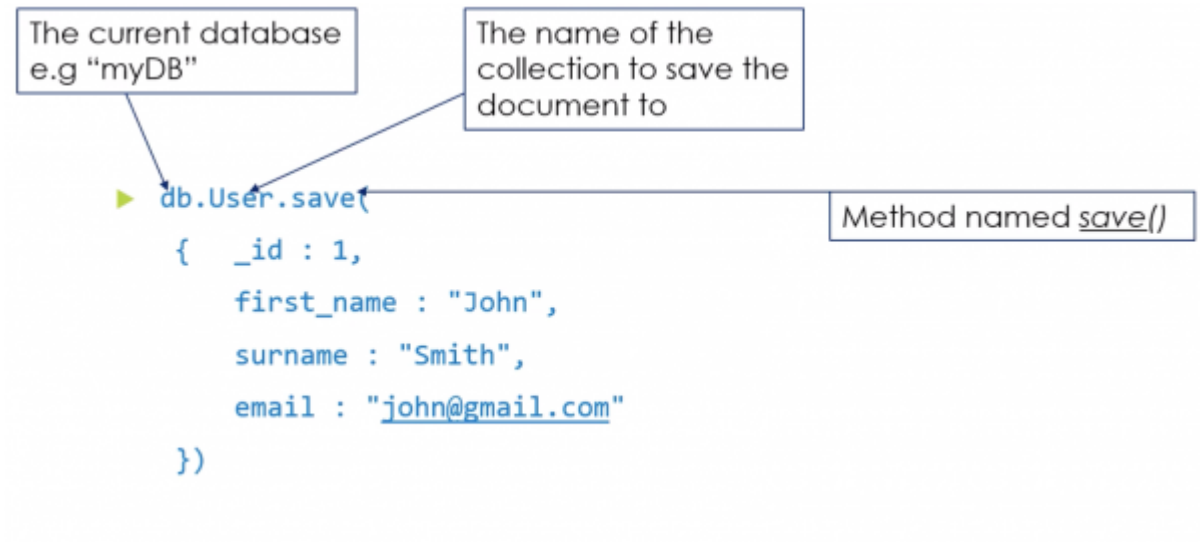
- show dbs - Show Databases
- use myDB - Switch to databases named “myDB” (If it doesn't exist, Mongo creates it)
- db - Show current Database.
- show collections - Show collections in the current database

MongoDB Rules for creating a Document

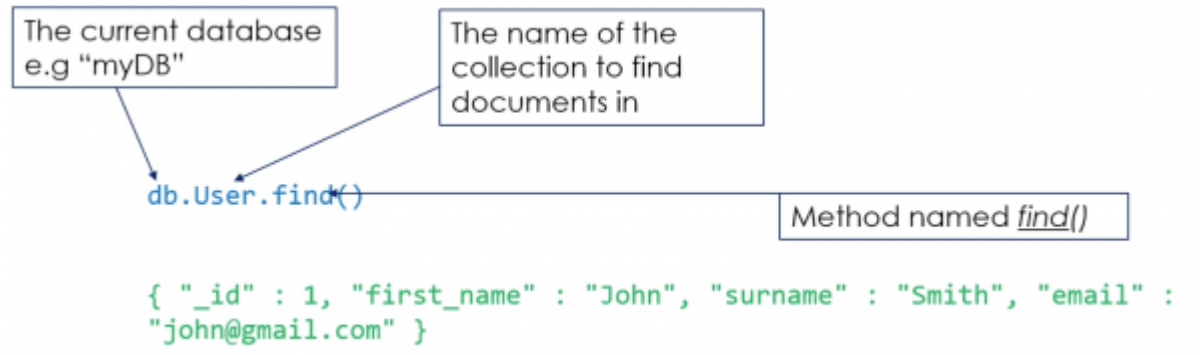
- Rules for MongoDB documents
    - A document must have an _id field. if one is not provided, it will be automatically generated

- The _id cannot be an array

Create a document - save()



Query the database - find()



pretty()

```
db.User.find().pretty()

{
        "_id" : 1,
        "first_name" : "John",
        "surname" : "Smith",
        "email" : "john@gmail.com"
}
```

```
db.User.find()
{ "_id" : 1, "first_name" : "John", "surname" : "Smith", "age" : 22, "email" : "john@gmail.com" }
{ "_id" : 2, "first_name" : "Sean", "surname" : "Williams", "age" : 30, "email" : "williamss@gmail.com" }
{ "_id" : 3, "first_name" : "Albert", "surname" : "O'Hara", "age" : 27, "email" : "al@hotmail.com", "twitter" : "@al1234" }
{ "_id" : 4, "first_name" : "Mary", "surname" : "Collins", "age" : 22 }
{ "_id" : 5, "first_name" : "Susan", "surname" : "Hanly", "age" : 18, "email" : "susie@hotmail.com", "twitter" : "@Susie2u" }
```

To find only documents where *age* is 22:

```
db.User.find({age:22})
```

```
{ "_id" : 1, "first_name" : "John", "surname" : "Smith", "age" : 22, "email" : "john@gmail.com" }
{ "_id" : 4, "first_name" : "Mary", "surname" : "Collins", "age" : 22 }
```

$and

To find only documents where *age* is 22 and *_id* is 1:

```
db.User.find({age:22, _id:1})
db.User.find({$and: [{age:22}, {_id:1}]})
```

```
{ "_id" : 1, "first_name" : "John", "surname" : "Smith", "age" : 22, "email" : "john@gmail.com" }
```

$or

To find only documents where *age* is 22 or 30

```
db.User.find({$or: [{age:22}, {age:30}]})
```

```
{ "_id" : 1, "first_name" : "John", "surname" : "Smith", "age" : 22, "email" : "john@gmail.com" }
{ "_id" : 2, "first_name" : "Sean", "surname" : "Williams", "age" : 30, "email" :
"williamss@gmail.com" }
{ "_id" : 4, "first_name" : "Mary", "surname" : "Collins", "age" : 22 }
```

$in

To find only documents where *age* is 22 or 30

```
db.User.find({age: {$in: [22, 30]}})
```

```
{ "_id" : 1, "first_name" : "John", "surname" : "Smith", "age" : 22, "email" : "john@gmail.com" }
{ "_id" : 2, "first_name" : "Sean", "surname" : "Williams", "age" : 30, "email" :
"williamss@gmail.com" }
{ "_id" : 4, "first_name" : "Mary", "surname" : "Collins", "age" : 22 }
```

Attribute

To find only documents that have a *twitter* attribute

```
db.User.find({twitter: {$exists:true}})
```

```
{ "_id" : 3, "first_name" : "Albert", "surname" : "O'Hara", "age" : 27, "email" : "al@hotmail.com",
"twitter" : "@al1234" }
{ "_id" : 5, "first_name" : "Susan", "surname" : "Hanly", "age" : 18, "email" :
"susie@hotmail.com", "twitter" : "@Susie2u" }
```

Attribute and age is greater than 20

To find only documents that have a *twitter* attribute and *age* is greater than 20

```
db.User.find({$and: [{twitter: {$exists: true}}, {age: {$gt: 20}}]})
```

```
{ "_id" : 3, "first_name" : "Albert", "surname" : "O'Hara", "age" : 27, "email" : "al@hotmail.com",
"twitter" : "@al1234" }
```

findOne()

To find only one document that has a *twitter* attribute

```
db.User.findOne({twitter: {$exists: true}})
```

```
{ "_id" : 3, "first_name" : "Albert", "surname" : "O'Hara", "age" : 27, "email" : "al@hotmail.com",
"twitter" : "@al1234" }
```

sort()

To sort all documents that have a *twitter* attribute alphabetically by *surname* and within *surname*, from oldest to youngest

```
db.User
.find({twitter: {$exists: true}}).sort({surname:1 , age:-1})
{ "_id" : 5, "first_name" : "Susan", "surname" : "Hanly", "age" : 18, "email" :
"susie@hotmail.com", "twitter" : "@Susie2u" }
{ "_id" : 3, "first_name" : "Albert", "surname" : "O'Hara", "age" : 27, "email" : "al@hotmail.com",
"twitter" : "@al1234" }
```

MongoDB -_id

- As previously described, the document ID (_id) attribute of a mongoDB document is the only mandatory part of a document.
- It can be any value, except an array.

```
db.Test.save({_id: 1, name: "John"})

db.Test.find({name: "John"})

{ "_id" : 1, "name" : "John" }
```

```
db.Test.save({name: "Billy"})

db.Test.find({name: "Billy"})

{ "_id" : ObjectId("591acc5612b8754878ffac81"),
  "name" : "Billy" }
```

more on save()

```
db.mydoc.find()
{ "_id" : 1, "name" : "Tom", "age" : 37 }
{ "_id" : 2, "name" : "Bill", "age" : 44 }
{ "_id" : 3, "name" : "Mary", "age" : 29 }
{ "_id" : 4, "name" : "Susan", "age" : 37 }

db.mydoc.save({_id:1, name:"John", age:24})

db.mydoc.find()

{ "_id" : 1, "name" : "John", "age" : 24 }
{ "_id" : 2, "name" : "Bill", "age" : 44 }
{ "_id" : 3, "name" : "Mary", "age" : 29 }
{ "_id" : 4, "name" : "Susan", "age" : 37 }
```

insert()

- Insert a document or documents into a collection.

```
db.mydoc.find()
{ "_id" : 1, "name" : "Tom", "age" : 37 }
{ "_id" : 2, "name" : "Bill", "age" : 44 }
{ "_id" : 3, "name" : "Mary", "age" : 29 }
{ "_id" : 4, "name" : "Susan", "age" : 37 }
db.mydoc.insert({_id: 1, name: "John", age: 24})
db.mydoc.find()

{ "_id" : 1, "name" : "Tom", "age" : 37 }
{ "_id" : 2, "name" : "Bill", "age" : 44 }
{ "_id" : 3, "name" : "Mary", "age" : 29 }
{ "_id" : 4, "name" : "Susan", "age" : 37 }
```

```
db.mydoc.find()
{ "_id" : 1, "name" : "Tom", "age" : 37 }
{ "_id" : 2, "name" : "Bill", "age" : 44 }
{ "_id" : 3, "name" : "Mary", "age" : 29 }
{ "_id" : 4, "name" : "Susan", "age" : 37 }
db.mydoc.insert([{_id: 5, name: "Sean", age: 54},{_id:6, name: "Luke"}])

db.mydoc.find()
{ "_id" : 1, "name" : "Tom", "age" : 37 }
{ "_id" : 2, "name" : "Bill", "age" : 44 }
{ "_id" : 3, "name" : "Mary", "age" : 29 }
{ "_id" : 4, "name" : "Susan", "age" : 37 }
{ "_id" : 5, "name" : "Sean", "age" : 54 }
{ "_id" : 6, "name" : "Luke" }
```

update()

- Modifies an existing document or docents in a collection
- Update (query, update, options)[2)]

Does not update Mary

```
db.mydoc.find()
{ "_id" : 1, "name" : "Tom", "age" : 37 }
{ "_id" : 2, "name" : "Bill", "age" : 44 }
{ "_id" : 3, "name" : "Mary", "age" : 29 }
{ "_id" : 4, "name" : "Susan", "age" : 37 }
db.mydoc.update( {$or: [{name:"Tom"}, {name:"Mary"}]}, {address: "Galway"})

db.mydoc.find()
{ "_id" : 1, "address" : "Galway" }
{ "_id" : 2, "name" : "Bill", "age" : 44 }
{ "_id" : 3, "name" : "Mary", "age" : 29 }
{ "_id" : 4, "name" : "Susan", "age" : 37 }
```

$set

```
db.mydoc.find()
{ "_id" : 1, "name" : "Tom", "age" : 37 }
{ "_id" : 2, "name" : "Bill", "age" : 44 }
{ "_id" : 3, "name" : "Mary", "age" : 29 }
{ "_id" : 4, "name" : "Susan", "age" : 37 }
```

```
db.mydoc.update( {$or: [{name:"Tom"}, {name:"Mary"}]}, {$set:{address: "Galway"}},  {multi:true})
```

```
db.mydoc.find()
{ "_id" : 1, "name" : "Tom", "age" : 37, "address" : "Galway" }
{ "_id" : 2, "name" : "Bill", "age" : 44 }
{ "_id" : 3, "name" : "Mary", "age" : 29, "address" : "Galway" }
{ "_id" : 4, "name" : "Susan", "age" : 37 }
```

```
db.emp.find()
{ "_id" : 1, "name" : "Tom", "experience" : 17 }
{ "_id" : 2, "name" : "Bill", "experience" : 3 }
{ "_id" : 3, "name" : "Mary", "experience" : 13 }
{ "_id" : 4, "name" : "Susan", "experience" : 5 }
```

```
db.mydoc.update( {experience: {$gt:20}}, {$set: {title:"Manager"}},  {multi:true, upsert:true})
```

```
db.mydoc.find()
{ "_id" : 1, "name" : "Tom", "experience" : 17 }
{ "_id" : 2, "name" : "Bill", "experience" : 3 }
{ "_id" : 3, "name" : "Mary", "experience" : 13 }
{ "_id" : 4, "name" : "Susan", "experience" : 5 }
{ "_id" : ObjectId("5c7bbf654be40b2777d5c006"), "title" : "Manager" }
```

deleteOne()

- Removes a single document from a collection

```
db.mydoc.find()
{ "_id" : 1, "name" : "Tom", "age" : 37 }
{ "_id" : 2, "name" : "Bill", "age" : 44 }
{ "_id" : 3, "name" : "Mary", "age" : 29 }
{ "_id" : 4, "name" : "Susan", "age" : 37 }
```

```
db.mydoc.deleteOne({age:{$lt:40}})
```

```
db.mydoc.find()
{ "_id" : 2, "name" : "Bill", "age" : 44 }
{ "_id" : 3, "name" : "Mary", "age" : 29 }
{ "_id" : 4, "name" : "Susan", "age" : 37 }
```

deleteMany()

```
db.mydoc.find()
{ "_id" : 1, "name" : "Tom", "age" : 37 }
{ "_id" : 2, "name" : "Bill", "age" : 44 }
{ "_id" : 3, "name" : "Mary", "age" : 29 }
{ "_id" : 4, "name" : "Susan", "age" : 37 }

db.mydoc.deleteMany({age:{$lt:40}})


db.mydoc.find()
{ "_id" : 2, "name" : "Bill", "age" : 44 }
```

Operators

https://docs.mongodb.com/manual/reference/operator/

Update Operators

https://docs.mongodb.com/manual/reference/operator/update/

Logical Query Operators

https://docs.mongodb.com/manual/reference/operator/query-logical/

Comparison Query Operators

https://docs.mongodb.com/manual/reference/operator/query-comparison/

Topic 7 - MongoDB II

More on find()

```
db.user.find()
```

```
{ "_id" : 1, "first_name" : "John", "surname" : "Smith", "age" : 22, "email" : "john@gmail.com" }
{ "_id" : 2, "first_name" : "Sean", "surname" : "Williams", "age" : 30, "email" : "williamss@gmail.com" }
{ "_id" : 3, "first_name" : "Albert", "surname" : "O'Hara", "age" : 27, "email" : "al@hotmail.com", "twitter" : "@al1234" }
{ "_id" : 4, "first_name" : "Mary", "surname" : "Collins", "age" : 22 }
{ "_id" : 5, "first_name" : "Susan", "surname" : "Hanly", "age" : 18, "email" : "susie@hotmail.com", "twitter" : "@Susie2u" }
```

To find only documents that have an email attribute and age is greater than 20

```
db.user.find({$and:[{email: {$exists:true}}, {age:{$gt:20}}]})
```

```
{ "_id" : 1, "first_name" : "John", "surname" : "Smith", "age" : 22, "email" : "john@gmail.com" }
{ "_id" : 2, "first_name" : "Sean", "surname" : "Williams", "age" : 30, "email" :
"williamss@gmail.com" }
{ "_id" : 3, "first_name" : "Albert", "surname" : "O'Hara", "age" : 27, "email" : "al@hotmail.com",
"twitter" : "@al1234" }
```

find(query, projection)

```
db.User.find()
```

```
{ "_id" : 1, "first_name" : "John", "surname" : "Smith", "age" : 22, "email" : "john@gmail.com" }
{ "_id" : 2, "first_name" : "Sean", "surname" : "Williams", "age" : 30, "email" : "williamss@gmail.com" }
{ "_id" : 3, "first_name" : "Albert", "surname" : "O'Hara", "age" : 27, "email" : "al@hotmail.com", "twitter" : "@al1234" }
{ "_id" : 4, "first_name" : "Mary", "surname" : "Collins", "age" : 22 }
{ "_id" : 5, "first_name" : "Susan", "surname" : "Hanly", "age" : 18, "email" : "susie@hotmail.com", "twitter" : "@Susie2u" }
```

Return only the email attribute of documents where age is greater than 18

```
db.User.find({age: {$gt: 20}}, {email:1})
```

```
{ "_id" : 1, "email" : "john@gmail.com" }
{ "_id" : 2, "email" : "williamss@gmail.com" }
{ "_id" : 3, "email" : "al@hotmail.com" }
{ "_id" : 4 }
```

Return only the first_name and surname attributes of all documents

```
db.User.find({}, {_id:false, first_name:1, surname:1})
```

```
{ "first_name" : "John", "surname" : "Smith" }
{ "first_name" : "Sean", "surname" : "Williams" }
{ "first_name" : "Albert", "surname" : "O'Hara" }
{ "first_name" : "Mary", "surname" : "Collins" }
{ "first name" : "Susan", "surname" : "Hanly" }
```

aggregate()

- Calculates aggregate values for the data in a collection
- db.collection.aggregate(pipeline, options)
  - pipeline stages
  - pipeline Operators

Example

```
{ "_id" : 1, "name" : "John", "age" : 23, "gpa" : 4.5, "sex" : "M" }
{ "_id" : 2, "name" : "Tom", "age" : 22, "gpa" : 3.5, "sex" : "M" }
{ "_id" : 3, "name" : "Mary", "age" : 24, "gpa" : 3.5, "sex" : "F" }
{ "_id" : 4, "name" : "Sarah", "age" : 22, "gpa" : 4, "sex" : "F" }
{ "_id" : 5, "name" : "Bill", "age" : 23, "gpa" : 3, "sex" : "M" }
```

Get the average gpa for all students

```
db.users.aggregate([{$group:{_id:null, Average{$avg:"$gpa"}}}])
```

$group same as Group by in MYSQL

Result:

```
{ "_id" : null, "Average" : 3.7 }
```

Get the Maximum GPA per age group

```
db.march8.aggregate([{$group:{_id:"$age", "Max GPA per Age":{$max:"$gpa"}}}])
```

```
{ "_id" : 24, "Max GPA per Age" : 3.5 }
{ "_id" : 22, "Max GPA per Age" : 4 }
{ "_id" : 23, "Max GPA per Age" : 4.5 }
```

To sort: $sort

```
db.march8.aggregate([{$group:{_id:"$age", "Max GPA per Age":{$max:"$gpa"}}}, {$sort:{_id:1}}])
```

```
{ "_id" : 22, "Max GPA per Age" : 4 }
{ "_id" : 23, "Max GPA per Age" : 4.5 }
{ "_id" : 24, "Max GPA per Age" : 3.5 }
```

Indexing

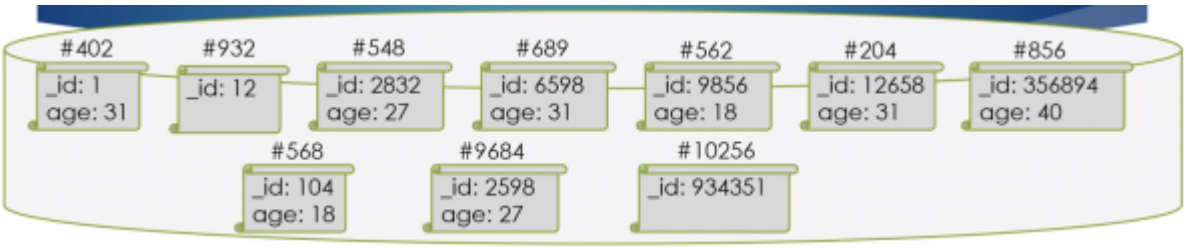
```
db.user.find()
```

Return all documents where age is greater than 18

Psuedo code example

```
for each document d in 'user'{
  if(d.age == 35){
    return d;
  }
}
```

- Indexes support the efficient execution of queries in MongoDB.
- Without indexes, MongoDB must perform a collection scan, i.e scan every document in a collection, to select those documents that match the query statement.
- Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form.
- Indexes hold mappings from field values to document locations.



age Index Table	
Value	Disk Location
18	#562, #568
27	#548, #9684
31	#402, #204, #689
40	#856

getIndexes()

- By default the only index on a document is on the _id field.
- To find the indexes on a collection:

```
db.collection.getIndexes()
```

Which returns information in the following format, detailing the index field (_id) and the order of the indexes(1 is ascending:-1 is descending):

```
"key":{

  "_id":1
}
```

createIndex()

- To create an index on a field other than **_id:**
- **db.collection.createIndex()**

```
db.User.find()
{ "_id" : 1, "first_name" : "John", "surname" : "Smith", "age" : 22, "email" : "john@gmail.com" }
{ "_id" : 2, "first_name" : "Sean", "surname" : "Williams", "age" : 30, "email" : "williamss@gmail.com" }
{ "_id" : 3, "first_name" : "Albert", "surname" : "O'Hara", "age" : 27, "email" : "al@hotmail.com", "twitter" : "@al1234" }
{ "_id" : 4, "first_name" : "Mary", "surname" : "Collins", "age" : 22 }
{ "_id" : 5, "first_name" : "Susan", "surname" : "Hanly", "age" : 18, "email" : "susie@hotmail.com", "twitter" : "@Susie2u" }
```

```
db.user.createIndex({age:1})
```

dropIndex()

- To drop an index on a field use:

```
db.collection.dropIndex()
```

- To drop the index on the age field we just created use:

```
db.collection.dropIndex({age:1})
```

- Note: The index on **_id** cannot be dropped

sort()

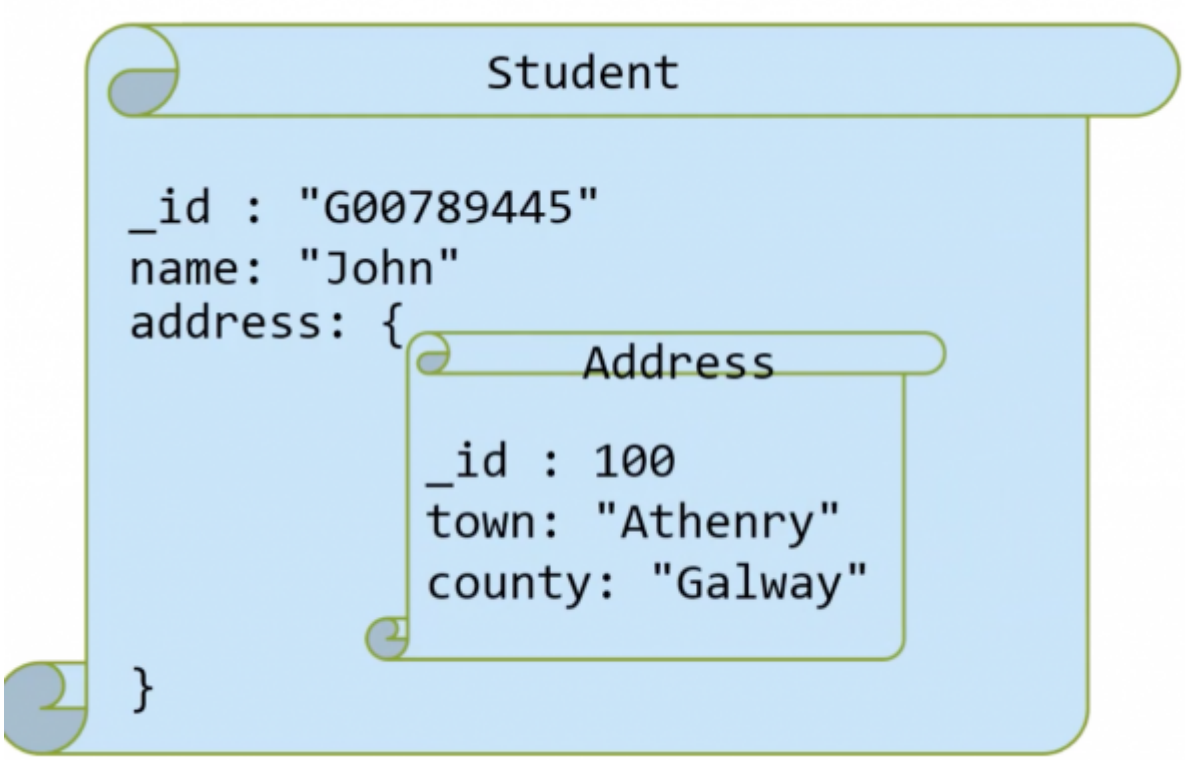
- When a sort() is performed on a field that is not an index, MongoDB will sort the results in memory.
- If the sort() method consumes more than 32MB of memory, MongoDB aborts the sort.
- To avoid this error, create an index supporting the sort operation.

__Relationships__ in MongoDB

- Modelling relationships between documents
  - One-to-One Relationships with Embedded Documents

- One-to-many Relationships with embedded Documents
- One-to-many relationships with document references

One-to-One relationships with embedded documents




```
db.student.save({_id:"G00789445",
                 name: "John",
                 address:{_id: 100,
                          town: "Athenry",
                          county:"Galway"}})
```

```
db.student.find({}, {address:1})
```

```
{ "_id" : "G00789445",
  "address" : {
    "_id" : 100,
    "town" : "Athenry",
    "county" : "Galway"
  }
}
```

- Show only the county field of documents that have an address field.

```
db.student.find({address:{$exists: true}}, {_id:0, "address.county":1})
```

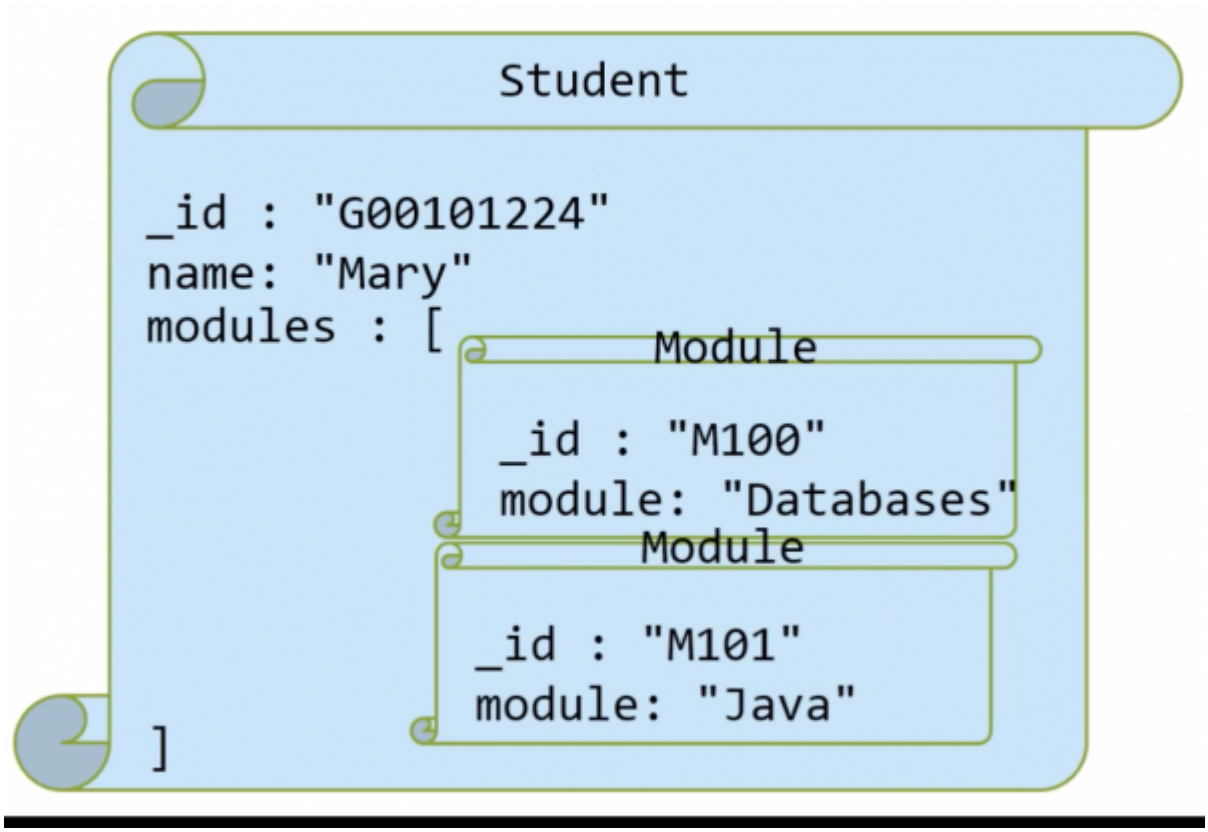
Note: Observe details in projection part of find, i.e. find(query, projection), {_id:0, "address.county":1}

{_id:0, "address.county":1}	Meaning
_id:0	Do NOT output **_id** field

{_id:0, “address.county”:1}	Meaning
“address.county”:1	Only output **county** field

```
{ "address" : { "county" : "Galway" } }
```

One-to-Many Relationships with Embedded Documents



Create the document with the relationships

```
db.student.save({_id:"G00101224",
                 name:"Mary",
                 modules:[{_id:"M100", module:"Databases"},
                          _id:"M101", module:"Java"}]})
```

Show the student's **_id** and **module** of all modules taken by student G00101224

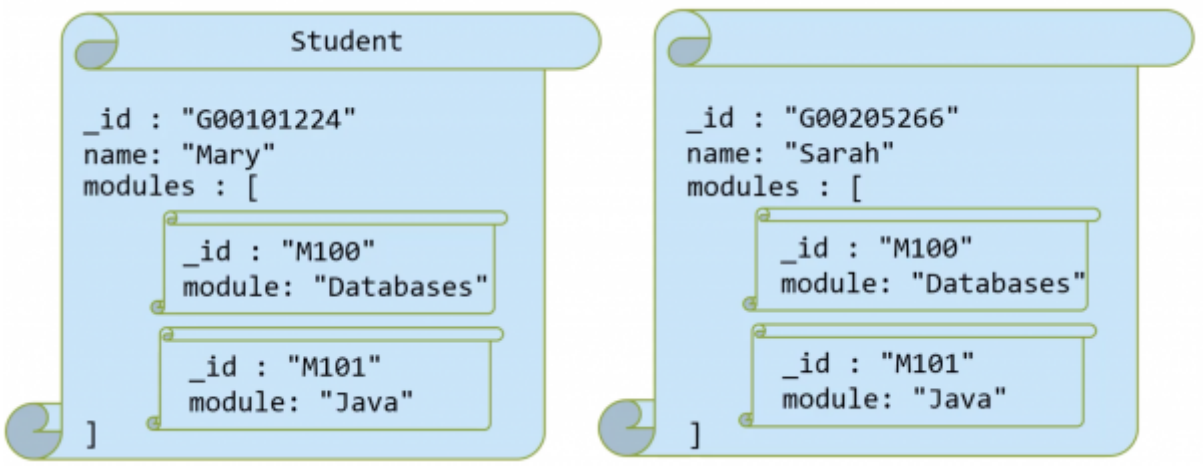
```
db.student.find({_id:"G00101224"}, {"modules.module":1})
```

projection - only show the **module** of the **modules field**

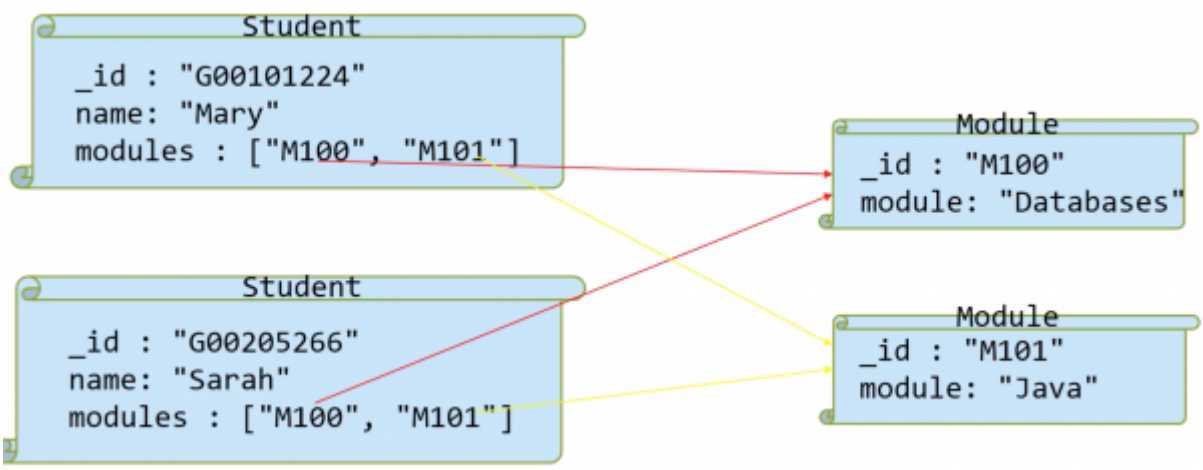
```
{"_id":"G00101224", "modules":[{"module":"Databases"},{"module":"Java"}]}
```

One-to-Many relationships with document References

In the example the document has only two field, but in reality it can be a very long document with much more information, so it makes sense to use relationships instead.



with referencing




```
//save the modules to the docs collection
db.docs.save({_id:"M100", module:"Databases"})
db.docs.save({_id:"M101", module:"Java"})
//save the students to the docs collection with references to the modules using the module _id
fields.
db.docs.save({_id:"G00101224", name:"Mary", modules["M100", "M101"]})
db.docs.save({_id:"G00205266", name:"Sarah", modules["M100", "M101"]})
```

$lookup

Using the $lookup *pipeline*... [3)]

Similar to a join in MySQL...

Performs a **left outer join** to an *unsharded* collection in the same database to filter in documents from the “joined” collection for processing. To each input document, the $lookup stage adds a new array field whose elements are the matching documents from the “joined” collection. The **$lookup** stage passes these reshaped documents to the next stage.

```
{
   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: <output array field>
     }
}
```

from - specifies the collection in the same databases to perform the join with. The from collection cannot be *sharded*.

localField - The value to search for.

foreignField - The field to search for the value specified by localField.

as - The name of the output.

```
{ "_id" : "M100", "module" : "Databases" }
{ "_id" : "M101", "module" : "Java" }
{ "_id" : "G00101224", "name" : "Mary", "modules" : [ "M100", "M101" ] }
{ "_id" : "G00205266", "name" : "Sarah", "modules" : [ "M101" ] }
```

Return all documents including the complete referenced documents

```
db.docs.aggregate([{$lookup:{from:"docs", localField:"modules", foreignField: "_id",
as:"Details"}}])
```

```
{ "_id" : "M100", "module" : "Databases", "Details" : [ ] }

{ "_id" : "M101", "module" : "Java", "Details" : [ ] }

{ "_id" : "G00101224", "name" : "Mary", "modules" : [ "M100", "M101" ],
"Details" : [ { "_id" : "M100", "module" : "Databases" }, { "_id" :"M101", "module" : "Java" } ] }

{ "_id" : "G00205266", "name" : "Sarah", "modules" : [ "M101" ],
"Details" : [ { "_id" : "M101", "module" : "Java" } ] }
```

Embedded Documents vs Referenced Documents

Features of embedded Documents

- Better performance
- Atomic

Features of Referenced Documents

- Slower
- No repetition
- More complex relationships

MongoDB vs MySQL

Features of MongoDB

- Huge amounts of data
- Unstructured
- Doesn't really support relationships

Features of MySQL

- Very Stable
- Structured
- Integrity

DATA ANALYTICS REFERENCE DOCUMENT	
Document Title:	Applied Databases - Python
Document No.:	1553629016
Author(s):	Gerhard van der Linde, Rita Raher
Contributor(s):	

REVISION HISTORY

Revision	Details of Modification(s)	Reason for modification	Date	By
0	Draft release	Applied Databases - Python	2019/03/26 19:36	Gerhard van der Linde

Topic8 - Python I

Databases vs Program

Employee ID	Name	Dept	Salary
100	John	HR	25500
101	Mary	R&D	44500
102	Bill	R&D	43000
103	Tom	Sales	40000

```
SELECT * from employees WHERE Salary > 42000;
SELECT * from employees WHERE Salary < 30000;
```

Variables

- Variables are names areas in the computer's memory that store values.

variables.py

```
        my1stVariable = "Hello World"
        my2ndVariable = 1

        print(my1stVariable)
        #Hello World

        my2ndVariable + 4
```

```
print(my2ndVariable)
# 1
```

- Variables are named areas in the computer's memory that store values.

variables2.py

```
my2ndVariable = 1
x = my2ndVariable + 4


print(my2ndVariable)
# 1
print(x)
# 5

age = 21
age = age + 1

print(age) # 22
```

IF Statements

ifstatements.py

```
age = 17

if(age > 17):
   print("OK")

print(finished)
# finsied
```

ifstatements1.py

```
age = 17

if(age > 17):
   print("OK")
elif(age < 18):
   print("Nok")
print(finished)
#Nok
# finished
```

ifstatements2.py

```
temp = 37

if(temp >37):
   print("Hot")
elif(temp <37):
   print("Cold")
else:
   print("OK")
print("Finished")
# ok
# finished
```

input

input.py

```
name = input("Enter name") # Tom
email = name + "@gmit.ie"
print(email)
# Tom@gmit.ie
```

salary.py

```
salary = input("Enter salary") # 30000
salary = int(salary)
salary = salary + 100

print(salary)
```

WHILE statement

while.py

```
i = 1
while(i <=5):
   print(i)
   i+=1
   # i = i +1
# 1
# 2
# 3
# 4
# 5
```

whilebreak.py

```
answer = "5"

while True:
   guess = input("Pick a number between 1 & 10")
   if(guess==answer):
     print("Correct!")
     break

print("end")
```

Arrays

array.py

```
myArr = ["Jan", "Feb", "March", "April"]
print(myArr)
#['Jan', 'Feb', 'March', 'April']

print(myArr[0])
# jan

print(len(myArr))
#4
```

Append()

append.py

```
myArr = ["Jan", "Feb", "March", "April"]
myArr.append("May")

print(myArr)
##['Jan', 'Feb', 'March', 'April', "May"]
```

FOR Statement

forloop.py

```
name = ["Tom", "John", "Mary", "Bob"]

for name in names:
   print(name + "@gmit.ie")

# Tom@gmit.ie
# John@gmit.ie
# Mary@gmit.ie
# Bob@gmit.ie


myArr = [1, 5, 12]

for x in myArr:
   print(x+1)
# 2
# 6
# 13

print(myArr)
#[1, 5, 12]
```

User-defined functions

userfunctions.py

```
def printMonths():
   print("Jan, Feb, Mar")

def printDays():
   print("Mon, Tue, Wed")

printDays()
# Mon, Tue, Wed

printMonths()
# Jan, Feb, Mar
```

name

userfunctions.py

```
def printMonths():
```

```
    print("Jan, Feb, Mar")

def main():
    printMonths()

if __name__ =="__main__":
    # execute only if run as a script
    main()
```

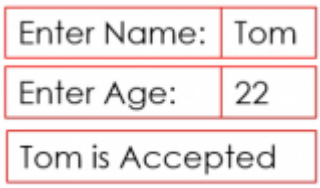
Parameters

parameters.py

```
print("Hello World")
# Hello World

print("Test")
# Test


s = "This is a string"

print(len(s))
# 16
```

parameters1.py

```
def checkAge(age):
    if age < 18:
        return "Too Young"
    return "Accepted"

def main():
    name = input("Enter:")
    age = int(input("Enter Age:"))
    print(name, "is", checkAge(age))

if __name__="__main__":
    # execute only if run as a script
    main()
```



Local Variables

A local variable is a variable that is given local scope. Local variable references in the function or block in which it is declared override the same variable name in the larger scope.

localvariables.py

```
def checkAge(age):
    limit = 18
    if age < limit:
        return "Too Young"
    return "Accepted"

def main():
```

```
    name = input("Enter:")
    age = int(input("Enter Age:"))
    print(name, "is", checkAge(age), limit)

if __name__="__main__":
    # execute only if run as a script
    main()
```

localvariables1.py

```
def checkAge(age):
    limit = 18
    if age < limit:
        return "Too Young"
    return "Accepted"

def main():
    limit= "Finished"
    name = input("Enter:")
    age = int(input("Enter Age:"))
    print(name, "is", checkAge(age), limit)

if __name__="__main__":
    # execute only if run as a script
    main()
```

Global Variables

globalvariables.py

```
def incrementAge(age):
    age += 1
    print(age)
    # 25

def main():
    age = 24
    incrementAge(age)
    print(age)
    # 24

if __name__="__main__":
    # execute only if run as a script
    main()
```

Declaring the variable outside of the function and using the keword global to make it a global variable

globalvariables.py

```
age = 24

def incrementAge(age):
    # access using the keyword "Global"
    global age
    age += 1
    print(age)
    # 25


def main():
    incrementAge(age)
    print(age)
    #25


if __name__="__main__":
```

```
        # execute only if run as a script
        main()
```

Topic9 - Python II

PyMySQL

- MySQLdb
- mysql.connector
- PyMySQL

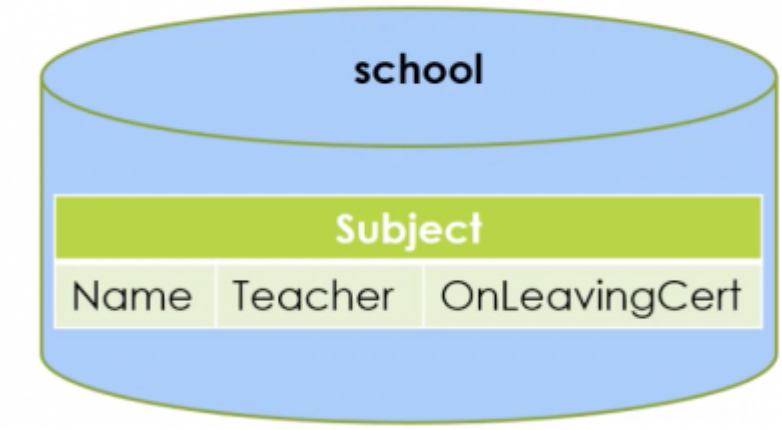
connect()

- The connect() function connects to a MySQL database.
- **host** - host where the database server is located
- **user** - username to log in as
- **password** - Password to use
- **db** - Database to use
- **port** - Port to use
- **cursorclass**- Custom cursor class to use

Connecting to the Database

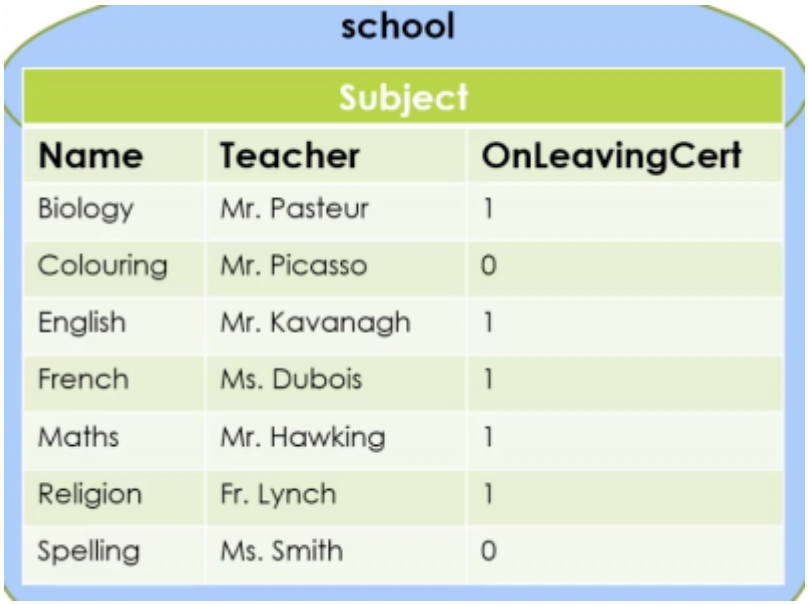
```
conn = pymysql.connect( "localhost", "root", "root", "school",
                cursorclass=pysql.cursors.DictCursor)
```

```
conn = pymysql.connect( "localhost", "root", "root", "school",
                cursorclass=pysql.cursors.DictCursor,
                password="root",
                host="localhost",
                db="school",
                port=3306)
```



Executing a query

```
query = "SELECT * FROM subject"

with conn:
   cursor = conn.cursor()
   cursor.execute(query)
   subjects = cursor.fetchall()
   for s in sujects:
      print(s["Name"])
```



school

Subject

Name	Teacher	OnLeavingCert
Biology	Mr. Pasteur	1
Colouring	Mr. Picasso	0
English	Mr. Kavanagh	1
French	Ms. Dubois	1
Maths	Mr. Hawking	1
Religion	Fr. Lynch	1
Spelling	Ms. Smith	0

query.py

```
      query = "SELECT * FROM subject
               WHERE teacher LIKE %s"

      with conn:
         cursor = conn.cursor()
         cursor.execute(query, ("Ms.%"))
         subjects = cursor.fetchall()
         for s in subjects:
            print(s["Name"])
```

Inserting new data

insertquery.py

```
      ins = "Insert INTO subject
             (Name, Teacher, OnLeavingCert)
             VALUE(%s, %s, %s)"

      with conn:
         cursor = conn.cursor()
         cursor.execute(ins, ("Maths", "Ms.Jones", 1))
         conn.commit()    # commit to database to make a change
```

try and except block for error messages

insertquery2.py

```
      ins = "Insert INTO subject
             (Name, Teacher, OnLeavingCert)
             VALUE(%s, %s, %s)"

      with conn:
         try:
            cursor = conn.cursor()
            cursor.execute(ins, ("Maths", "Ms.Jones", 1))
            conn.commit()
            print("Insert successful")
         except:
            print("Maths already exists")
```

Exceptions

```
name = "Maths"
teacher = "Ms.Jones"
lc =1
with conn:
   try:
      cursor = conn.cursor()
      cursor.execute(query, (name, teacher, lc))
      conn.commit()
      print("Insert Successful")
    except pymysql.err.InternalError as e:
      print("Internal Error", e)
    except pymysql.err.IntegrityError as e:
      print("Error", name, "already exists")
    except Exception as e:
      print("error", e)
```

Deleting Data

```
query = "DELETE FROM subject WHERE name = %s"
name = "Maths"

with conn:
   try:
      cursor = conn.cursor()
      rowAffected = cursor.execute(query, (name))
      conn.commit()
      if(rowsAffected ==0):
         print("Nothing deleted - ", name , "never existed")
      else:
         print(rowAffected, "row(s) deleted")
      except Exception as e:
         print("error", e)
```

Updating Data

```
query = "UPDATE subject SET teacher = %s WHERE NAME = %s"
subject = "Maths"
newTeacher = "Mr.Murphy"

with conn:
   try:
      cursor = conn.cursor()
      rowsAffected = cursor.execute(query, (newTeacher, subject))
```

```
        conn.commit()
        if(rowsAffected ==0):
            print(subject, "not updated")
        else:
            print(subject, "now taught by", newTeacher)
    except Exception as e:
        print("error", e)
```

Installing PyMySQL

- open command prompt or terminal
- type conda install pymysql
- create a new py file

Topic10 - Python III

topic_10_-_python_iii.pdf

pymongo

- client = pymongo.MongoClient()
- client = pymongo.MongoClient(host="localhost", port="27017")
- try
    - client.admin.command('ismaster')

Database and Collections

- mydb = myclient["cars"]
- cols = mydb.list_collection_names()
- docs = mydb["docs"]

find()

- people = docs.find({"age":{"$gt":18}})
- for person in people:
    - print(person["Name"])

find()

- people = docs.find({"age":{"$gt":18}}, {"_id":0})
- people = docs.find({"age":{"$gt":18}}, limit=2)

insert_one()

- newDoc = {“_id”:991, “name”:“John”, “age”:44}
- mycol.insert_one(doc)

insert_many()

- newDocs = [{“_id”:991, “name”:“John”, “age”:44},
  - {“_id”:992, “name”:“Mary”, “age”:24},
  - {“_id”:992, “name”:“Mary”, “age”:35}]
- mycol.insert_many(newDocs)

Exceptions

- pymongo.errors.ConnectionFailure
- pymongo.errors.DuplicateKeyError
- newDocs = [{“_id”:991, “name”:“John”, “age”:44},
  - {“_id”:992, “name”:“Mary”, “age”:24},
  - {“_id”:992, “name”:“Mary”, “age”:35}]
- mycol.insert_many(newDocs) mycol.insert_many(newDocs, ordered=False)

delete_one()

- filter = {“age”:{“$gt”:44}}
- mycol.delete_one(filter)
- mycol.delete_one({“age”:{“$gt”:44}})

delete_many()

- filter = {“age”:{“$gt”:44}}
- result = mycol.delete_many(filter)
- DeleteResult
- print(result.deleted_count)

update_one()

- filter = {“age”:{“$gt”:44}}
- update = {“$inc”:{“age”:1}}
- mycol.update_one(filter, update)

update_many()

- filter = {“age”:{“$gt”:44}}
- update = {“$inc”:{“age”:1}}
- result = mycol.update_many(filter, update)
- UpdateResult
- print(result.modified_count)

Review

- MySQL
- MongoDB
- Python

1)

Sharding is a type of database partitioning that separates very large databases the into smaller, faster, more easily managed parts called data shards. The word shard means a small part of a whole.

2)

https://docs.mongodb.com/manual/reference/method/db.collection.update/#db.collection.update

3)

https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/